12-15-2016

A High Level Framework for Solver Independent Model Manipulation and Generation of Hybrid Solvers

Daniel Fontaine
University of Connecticut, Storrs CT, daniel.fontaine@gmail.com

A High Level Framework for Solver Independent Model Manipulation and Generation of Hybrid Solvers

Daniel Fontaine, PhD

University of Connecticut, 2016

Many critical real world problems, including problems in areas such as logistics, routing and scheduling are very difficult to solve computationally (often NP-hard). Various programming and algorithmic paradigms have been developed to deal with these problems, including Constraint Programming (CP), Integer Programming (IP) and Local Search (LS). These technologies are largely declarative in nature and rely on vastly different underlying mathematical and algorithmic approaches. Hence, each paradigm has inherent strengths and weaknesses making it more or less suitable to a given problem. For particularly difficult problems, it can often be beneficial to leverage a sophisticated hybrid solver technique. Such techniques include, for example, combining CP and IP solvers in a cooperative fashion or iteratively solving problem relaxations such as in Lagrangian Relaxation, Column Generation or Logic-Based Benders Decomposition. The development of such hybrids, however, is often technically difficult and requires a great deal of trial and error. This thesis introduces a new high-level framework for automating the generation of several important classes of hybrid solvers as well as proposing a new set of theoretical abstractions allowing high-level model descriptions to be transformed and combined into hybrids while maintaining semantic soundness. Among the new theoretical abstractions is a proposal for 'Generic Lagrangian Relaxation', allowing a well-known Integer Programming technique to be generalized and applied to other technologies such as CP. Experimental results demonstrate the practical benefits of this new framework.

A High Level Framework for Solver Independent Model Manipulation and Generation of Hybrid Solvers

Daniel Fontaine

B.S., University of Connecticut, 2005

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

A High Level Framework for Solver Independent Model Manipulation and Generation of Hybrid Solvers

Presented by

Daniel Fontaine, B.S.

Major Advisor _____

Laurent Michel

Associate Advisor _____

Alexander Russell

Associate Advisor _____

Alexander Schwarzmann

University of Connecticut

2016

Credits

This dissertation incorporates research results appearing in the following publications:

- Daniel Fontaine and Laurent Michel. A high level language for solver independent model manipulation and generation of hybrid solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, volume 7298 of *Lecture Notes in Computer Science*, pages 180–194. Springer Berlin Heidelberg, 2012

- Daniel Fontaine, LaurentMichel, and Pascal Van Hentenryck. *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, chapter Constraint-Based Lagrangian Relaxation, pages 324–339. Springer International Publishing, Cham, 2014

- Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings*, chapter Parallel Composition of Scheduling Solvers, pages 159–169. Springer International Publishing, Cham, 2016

- Laurent Michel Daniel Fontaine. A large-scale neighborhood search approach to matrix decomposition into consecutive-ones matrices. 8th Workshop on Local Search techniques in Constraint Satisfaction, 9 2011

- Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. Model combinators for hybrid optimization. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 299–314. Springer Berlin Heidelberg, 2013

Acknowledgements

First, I would like to thank my family for supporting me through the long hours required to complete my degree.

I would also like to thank my advisor, Laurent Michel, who is the hardest working person I know, but always makes time for his students. Over the last several years he has been an excellent mentor and an inspiration.

Finally, I would like to thank my doctoral committee, Laurent Michel, Alexander Russell and Alexander Schwarzmann, for the time and effort they have put into reviewing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Discrete Optimization

Discrete Optimization problems play a central and ever increasing role in the modern world. Industries from telecommunications and microchip design to medicine and logistics are constantly grappling with fundamentally difficult *discrete optimization* problems. The core difficulty of these problems is that they are often combinatorial in nature with an exponential space of potential solutions. Finding the best solution, or in some cases, even a feasible solution can be computationally intractable even for the most powerful computers. Over the past few decades, numerous techniques have been developed to grapple with these categories of problems. The most well-known and studied technique has been Integer Programming (IP) with its origins going back to the 1960s. In the past couple decades, however, numerous other techniques have been developed including Boolean Satisfiability (SAT), Constraint Programming (CP), and numerous Local Search (LS) methods. Despite progress, finding satisfactory solutions to these problems remains a daunting task. Beyond the primary computational difficulties presented by these problems, a myriad of secondary difficulties also arise in practice. In particular, among the available solver technologies, there is no clear 'best choice' a priori. The most effective technology will vary from problem to problem and in

many cases even from instance to instance of the same problem. Furthermore, once a solver technology has been chosen, the tractability of the problem is typically highly sensitive to the way in which the problem is modeled. That is, the set of decision variables, constraints and objective functions that are chosen to describe the problem. Once a suitable model has been formulated, an effective search procedure must also be written or selected from among pre-existing black-box search heuristics. The end-result is a highly fragile process often requiring years of experience and specialized knowledge to be effective.

In recent years, the situation has gotten worse still, as many modern techniques are *hybrid* approaches relying on more than one solver technology or problem formulation. Such *hybrid* solvers must glue together disparate optimization techniques providing communication between solvers and possibly transcoding of solutions between different models. Hybrid solvers are notoriously challenging to write and maintain with many subtleties that must be carefully considered. The major contribution of this thesis is the development of a high-level framework based on new theoretical abstractions that substantially reduce the burden of building sophisticated hybrids and allows problems to be easily transformed for use with multiple solver technologies or hybrid approach. In particular, suitable high-level abstractions allow:

1. Compact and expressive modeling of the problem.
2. Easy translation to a low-level, technology-specific solver.
3. Implicit transcoding of solutions between solver technologies.
4. Combining solvers to run and communicate in parallel or sequentially.
5. Easy application of common algorithmic templates to a model.
6. Semantic validation of hybrid solvers.

The remainder of this chapter will introduce the formalisms used for *constraint satisfaction problems* (CSP) and *constraint optimization problems* (COP), provide some examples

of *constraint optimization problems* and then introduce technologies commonly used to solve these problems.

1.2 Definition & Examples of Constraint Optimization Problems

An *optimization problem* may be either *discrete* or *continuous* indicating whether the problem's decision variables have discrete or continuous domains. Algorithms for solving *continuous optimization problems* often leverage numerical techniques familiar from calculus or linear algebra allowing for effective polynomial-time optimization. *Discrete Optimization*, the focus of this thesis, requires decision variables to be assigned integer values only. Hence, the analytical techniques used in the *continuous* case aren't directly applicable to *discrete optimization problems*. Although *discrete optimization problems* can fall into numerous *computational complexity classes*, many problems of both theoretical and practical importance are *NP-hard* with the best known algorithmic approaches relying on branch-and-bound techniques on exponential-sized search spaces.

Discrete combinatorial problems can be captured explicitly in the form of *constraint satisfaction problems* (CSP) or *constraint optimization problems* (COP). These concepts can be described formally as follows:

Definition 1 (constraint satisfaction problems(CSP)) *A CSP is of the form* $\langle X, D, C \rangle$ *where* X *is a set of decision variables,* D *is a set of nonempty discrete domains associated with* X*, and* C *is a set of constraints over* X *defining the solution space of the problem. Each variable,* $x_i \in X$*, may be assigned a value from the corresponding domain* $d_i \in D$ *(also denoted* $D(x_i)$*). Each constraint* $c_j \in C$ *is a pair* $\langle Y_j, R_j \rangle$*, where* $Y_j \subseteq X$ *and* R_j *is a relation over the variables* Y_j*. The constraint* c_j *is satisfied when the values assigned to variables* Y_j *satisfy the relation* R_j*.*

Definition 2 (variable assignment) *A variable assignment is a set* $\sigma_X = \{(x, v)\ s.t.\ v \in D(x)\}$ *and each* $x \in X$ *appears exactly once. An assignment associates a value to each variable from its domain.*

Definition 3 (Solution (feasible assignment)) *A variable assignment, denoted* σ_X*, is feasible if it satisfies all the constraints in C, that is* $c_1(\sigma_X) \wedge \ldots \wedge c_k(\sigma_X)$ *(where* $k = |C|$*). Such a feasible assignment is called a solution to the CSP.*

There are many examples of CSPs, including well known puzzles such as *sudoku*, *n-queens*, and *magic series* (for examples see a textbook such as [9]). Notice that a CSP is not an *optimization problem* as it has no notion of a solution's fitness. That is, all solutions to a CSP are equally good. The definition of a CSP may be expanded to that of a *constraint optimization problem* (COP) by adding an *objective function*.

Definition 4 (constraint optimization problems (COP)) *A COP is a CSP of the form* $\langle X, D, C, f\rangle$ *where* f *is an objective function to be minimized. The objective function,* $f : \Sigma \rightarrow \mathcal{R}$*, maps the space of variable assignments into the real numbers providing a fitness measure for a given variable assignment. Note that it may be desirable to maximize an objective function, but such a case can be formulated as a minimization of* $-f(x)$*, hence all optimization problems may be formulated as minimization problems without any loss of generality.*

A COP may be written in the following general form:

$$\mathcal{M} = \min_x f(x)$$

subject to

$$\begin{cases} C_1(x) \\ \vdots \\ C_k(x) \end{cases}$$

Here, x is the set of decision variables, $C_1 \ldots C_k$ are the constraints and the objective is to minimize the objective function f. A solver for $\mathcal{M}$ is a computer program which enumerates (explicitly or implicitly) the solution space $\mathcal{S}(\mathcal{M})$ trying to find all feasible solutions or one global optimal solution. Techniques for solving COPs fall into two categories, *complete* and *incomplete*. *Complete* COP methods are those designed to not only provide an optimal solution, but also prove that no better solution could be found in the search space. The optimality proof typically comes in the form of a search procedure which explicitly explores some portion of the search space while implicitly eliminating the rest of the space from consideration. *Incomplete* COP methods are those that may provide very good or even optimal solutions, but aren't designed to to prove that better solutions do not exist. It is important to note that a *complete search* algorithm is often impractical to run to completion for many combinatorial problems as the search space is simply too large.

1.2.1 Example: Assignment Problem

The *assignment problem* (see [16]) is a simple problem to formulate, yet is NP-hard to solve to optimality. The problem consists of assigning *agents* to *tasks* where each *agent-task* pair has a corresponding cost. The objective is to minimize the total cost of assigning exactly one *agent* to each *task*. Assume we have n agents each capable of performing one of m tasks and a cost matrix, C, of size $n \times m$ capturing the cost of each potential assignment. We may model the *assignment problem* with a vector of decision variables, a, of size m with domains in the range $1..n$ representing task assignments.

$$\mathcal{M} = \min \sum_{t=1}^{m} C_{a_t,t}$$

subject to

$$\left\{ \; alldifferent(a)$$

Here, *alldifferent* is a *global constraint* indicating that variables in a must be pairwise distinct. The objective is to minimize the cost of our assignments, $\sum_{t=1}^{m} C_{a_t,t}$. Although

this problem is quite simple, *greedy algorithms* will generally fail to find an *optimal solution*. This problem will be further explored in chapter 5.

1.2.2 Example: Graph Coloring Problem

Graph coloring (covered in textbook [92]) is another relatively simple problem to formulate, yet has both practical and theoretical importance. The aim is to minimize the number of colors necessary to color a graph so that no two adjacent vertices have the same color. The following snippet is a typical formulation of the problem:

$$
\begin{aligned}
Z = \texttt{min}\ \ & m \\
\text{subject to} & \begin{cases}
v_i \leq m, & i \in 1..|V| \\
v_i \neq v_j, & (i,j) \in E \\
v_i \in 1..|V|, & i \in 1..|V| \\
m \in 1..|V|
\end{cases}
\end{aligned}
$$

Here, V is the set of vertices, E the set of edges, m a decision variable for the number of colors used and $\{v_i\}_{i \in 1..|V|}$ are decision variables representing the color assigned to the i-th vertex of V. Notice that the domain of each vertex-color variable is $\{1 \ldots |V|\}$ indicating that the problem will use at most $|V|$ colors (in the case of a complete graph for example). The objective is to minimize the variable m, which is constrained to be at least as large as the number of colors in use. Graph Coloring has many real world applications, including pattern matching, sports scheduling, and register allocation in compiler design. This problem will be solved using a hybrid in chapter 8.

1.2.3 Example: Warehouse Location Problem

The Warehouse Location Problem (WLP, see textbook [90]) considers the problem of opening new warehouse locations to supply existing stores. Each potential warehouse-store pair has an associated cost and each warehouse has a maximum capacity of stores that it can supply.

Each store must be supplied by at least one warehouse and the objective is to minimize the total cost. The total cost is defined to be the cost of supplying each store plus the fixed cost of opening the warehouses.

$$\mathcal{M} = \min \sum_{s \in \text{Stores}} \text{cost}_s + \sum_{w \in \text{Warehouses}} \text{fixed}_w * \text{open}_w$$

subject to

$$\begin{cases} \text{cost}_s = C_{s,\text{support}_s}, & \forall s \in \text{Stores} \\ \text{open}_{\text{support}_s} = 1, & \forall s \in \text{Stores} \\ \sum_{s \in \text{Stores}} (\text{support}_s = w) \leq \text{capacity}_w, & \forall w \in \text{Warehouses} \end{cases}$$

In the above *model*, we assume domains for the stores ($Stores = 1 \ldots n$) and warehouses ($Warehouses = 1 \ldots m$) have been defined as well as a cost matrix C for the warehouse-store pairs and a vector of *fixed costs* for opening each warehouse. Three variable vectors are introduced:

cost has dimensions *Stores* and a domain of $\{min(C) \ldots max(C)\}$. These variables capture the costs of supplying each of the stores.

open is a Boolean variable vector of size *Warehouses*. This array captures which of the warehouses are open.

support has dimensions *Stores* and a domain of *Warehouses* and captures the mapping of stores to the warehouses that supply them.

This problem will be used as a recurring example in this chapter was well as chapter 5.

1.2.4 Example: Jobshop Scheduling

Scheduling problems are a major area or interest in combinatorial optimization. One of the best known scheduling problems is the jobshop problem (covered in textbook [92]). An $n \times m$ jobshop problem has a set of n jobs (J) that must be scheduled on a set of m machines (M).

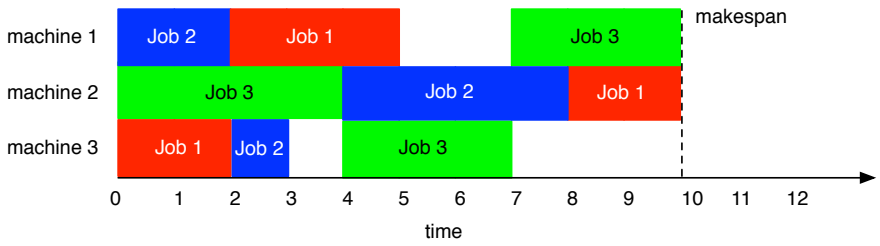

Figure 1.1: Example solution to a 3x3 jobshop instance.

Each job $j \in J$ is a sequence of m tasks σ_i^j of duration $p_{i,j}$ to be scheduled on the machines. Namely, $(\sigma_1^j \dots \sigma_m^j)$ is the sequence of tasks for job j. Machines are disjunctive, meaning that at most one task can execute at any point in time. The model below uses global constraints and is adapted from [50].

$$\mathcal{M} = \texttt{min} \;\; makespan$$
$$\texttt{s.t.} \begin{cases} precedes(task_{i,j}, task_{i+1,j}) & \forall i \in M, \forall j \in J \\ finished\_by(task_{m,j}, makespan) & \forall j \in J \\ disjunctive(\{task_{\sigma_k^j,j} \mid j \in J, k \in 1 \dots m, \sigma_k^j = r\}) & \forall r \in M \end{cases}$$

The objective is to minimize the *makespan*, a variable that captures the total time to complete the processing of all tasks. Figure 1.1 shows an example solution to a 3x3 jobshop instance. Notice that each particular job can have different processing times on different machines and that a job cannot be scheduled concurrently on two different machines. Jobshop scheduling hybrids will be demonstrated in Chapter 6.

1.2.5 Example: Intensity Modulated Radiation Therapy

This slightly more complicated example has real world applications in the medical field. The goal of IMRT (see [10]) is to deliver a prescribed dose of radiation to diseased tissue while minimizing exposure to vital organs and surrounding healthy tissue. One of the mechanisms for achieving this goal is 'shaping' the beam coming from the radiation source using

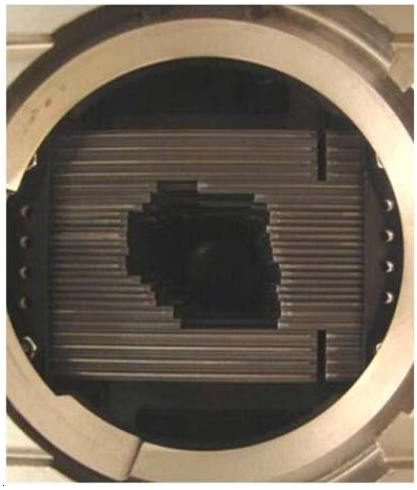

Figure 1.2: Multileaf Collimator (26 leaves)

a multileaf collimator.

Practitioners generate radiation intensity maps describing the desired distribution of radiation across the source beam. An intensity map can be described by a non-negative integer intensity matrix with entries indicating the relative prescribed dosage of radiation. The intensity matrix can then be administered using a multileaf collimator (picture in Figure 1.2). The collimator consists of pairs of leaves that can be moved into the beam field from the left and right. Regions of the beam occluded by collimator leaves will not receive radiation while those regions left exposed will. By irradiating a set of collimator configurations for varying lengths of time, the radiation profile specified by the intensity matrix may be realized.

The problem is to find a set of collimator configurations and associated *beam on* times that yield the desired intensity matrix. This problem may be recognized as the problem of decomposing the intensity matrix into a set of *consecutive ones* (C1) matrices. A C1 matrix is a binary matrix with the property that all ones in a row are consecutive. More formally, a C1 matrix satisfies:

$$(X_{k,i,L} = 1) \wedge (X_{k,i,R} = 1) \rightarrow (X_{k,i,M} = 1)$$

$$1 \leq L < M < R \leq n, \quad i \in 1..m$$

$$\begin{pmatrix} 0&4&6&7&3&3\\ 3&2&6&6&6&6\\ 0&1&1&1&2&0\\ 0&2&6&6&2&0\\ 1&5&4&6&4&4\\ 0&4&4&2&2&2 \end{pmatrix} = 1\begin{pmatrix} 0&0&0&1&1&1\\ 1&0&0&0&0&0\\ 0&1&1&1&0&0\\ 0&0&0&0&0&0\\ 1&1&0&0&0&0\\ 0&0&0&0&0&0 \end{pmatrix} + 2\begin{pmatrix} 0&0&1&1&1&1\\ 1&1&1&1&1&1\\ 0&0&0&0&1&0\\ 0&1&1&1&1&0\\ 0&0&0&1&0&0\\ 0&0&0&1&1&1 \end{pmatrix} + 4\begin{pmatrix} 0&1&1&1&0&0\\ 0&0&1&1&1&1\\ 0&0&0&0&0&0\\ 0&0&1&1&0&0\\ 0&1&1&1&1&1\\ 0&1&1&0&0&0 \end{pmatrix}$$

Figure 1.3: Decomposition of a 6×6 matrix into C1 matrices.

Thus, given an $m \times n$ intensity matrix I, we may formulate the problem as follows:

$$I = \sum_{k \in \Omega} b_k X_k$$

where Ω is an indexing set over the number of C1 matrices in the decomposition. The total *beam on* time is given by $B = \sum_{k \in \Omega} b_k$ and the *cardinality* is defined as $K = |\Omega|$. The quality of a decomposition is evaluated on the basis of a lexicographical minimization of B and then K, denoted *lex_min(B, K)*. Straightforward formulas exist for minimizing B, but the remaining problem of finding a minimal value of K given a minimal B remains NP-hard [10]. The difficulty of the problem grows not just with the size of the intensity matrix, but also with the domain of the matrix entries. It is common to denote the size of an intensity matrix instance in the following format, $n \times m \times k$ where n and m are the rows and columns and $\{0 \dots k\}$ is the domain of the entries in the matrix.

Figure 1.3 provides an example C1 decomposition of a $6 \times 6 \times 7$ intensity matrix. In this example, the total *beam on* time $B^* = 1 + 2 + 4 = 7$ and the cardinality $K = 3$.

Various modeling approaches to the radiation problem have been proposed in recent years and among the most successful has been the Counter Model [10]. The *Counter Model* is a novel model which counts the number of patterns according to their *beam on* time. Non-negative integer variables $Q_{b,i,j}$ represent the number of patterns with *beam on* time b that expose element $(i,\ j)$. These variables can be collected into matrices denoted Q_b. A C1 decomposition of I can be achieved by an unweighted C1 decomposition of the set of Q_b matrices. Since we wish to minimize the cardinality of our decomposition, we introduce non-negative integer variables N_b which represent the cardinality of a minimal decomposition of

matrix Q_b. We will precompute the optimal *beam on* time and denote it with the constant B^*. This can be found using the following simple formula:

$$B^* = max_{i \in 1..m} \sum_{j=1}^{n} inc(I_{i,j-1}, I_{i,j})$$

where $inc(I_{i,j-1}, I_{i,j}) = max(I_{i,j} - I_{i,j-1}, 0)$ and $I_{i,0} = 0, \forall i \in 1..m$. For greater insight into this formula, see [17]. The model also requires an upper bound on the beam intensity. A suitable upper bound may be found by taking the maximum value in the intensity matrix, $\bar{b} = max_{i \in 1..m, j \in 1..n} I_{i,j}$. The model may now be stated as follows:

minimize K

subject to:

$$\sum_{b=1}^{\bar{b}} bQ_{b,i,j} = I_{i,j}, \quad \forall i \in 1..m, j \in 1..n$$

$$N_b \geq \sum_{j=1}^{n} inc(Q_{b,i,j-1}, Q_{b,i,j}), \quad \forall i \in 1..m$$

$$K = \sum_{b=1}^{\bar{b}} N_b$$

$$B^* = \sum_{b=1}^{\bar{b}} bN_b$$

This model presents a *real world* application and demonstrates the thought and skill that go into compactly capturing such a problem. This model will be explored further in chapter 7.

1.3 Constraint Programming

Constraint Programming (CP) (refer to [76], [27], [9], [86], [87]) is among the newer technologies developed to address *combinatorial optimization problems* emerging as an independent

area of research in the 90's. Constraint Programming leverages a branch-and-prune tree search alongside inference-based domain filtering. Hence, CP brings a fundamentally different algorithmic approach than older techniques based on optimization techniques from algebra and calculus (such as Linear Programming and Integer Programming). CP, by design, is the most expressive and compact technology for modeling *optimization problems* as it makes use of non-linear constraints and *global constraints*, as well as non-linear objective functions. *Constraint Programming* also permits users to provide a custom search procedure or choose from a variety of 'blackbox' search heuristics. This section provides a high-level overview of CP as a solver technology, for a more in-depth understanding refer to textbooks such as [27] or [9].

1.3.1 Global Constraints

Optimization technologies such as *Integer Programming* (IP) and *Boolean Satisfiability* (SAT) require constraints to be expressed in a restricted form determined by the requirements of the solver algorithm. In the case of IP, all constraints must be linear inequalities. In SAT, constraints are Boolean clauses. *Global Constraints* (see [11], [42], [71]) are a more generic class intended to capture global combinatorial structures within a problem directly rather than requiring such structures to be 'encoded' using a limited class of constraints. The strength of this approach is two-fold:

1. CP allows a user to think about a problem at a much higher-level and to express it more compactly and intuitively.

2. Capturing global structures directly (as opposed to implicitly through an encoding) allows CP to leverage powerful domain filtering algorithms to dramatically reduce the search space.

To make this point concrete, consider *alldifferent* (see [42]), among the most used and studied *global constraints*. *Alldifferent* operates on a set of discrete variables (say $x_1 \dots x_n$)

and requires that the values of these variables are pairwise distinct ($x_i \neq x_j, \forall i, j \in \{1 \dots n\}, i \neq j$). Notice that this constraint captures a natural, yet non-linear, combinatorial structure. In most CP solvers, specifying this constraint is as simple as:

$$\text{alldifferent}(x_1, \dots, x_n)$$

To demonstrate the the advantage of leveraging combinatorial structure within *alldifferent*, consider the following CSP:

$$x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{2, 3\}$$
$$\text{alldifferent}(x_1, x_2, x_3)$$

Notice that both x_1 and x_2 have the domain $\{1, 2\}$ implying that one of these variables must be assigned the value 1 and the other the value 2. It follows that it is impossible to assign $x_3 = 2$ as this would leave the CSP infeasible. It is therefore safe to remove the value 3 from the domain of x_3. The *alldifferent* constraint is capable of detecting this unsupported value and pruning it. If, instead, a set of pairwise disjunctive constraints were used to model this CSP:

$$x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3$$

the overall structure is lost. Considering the above constraints individually, it is not possible to infer that $D(x_3)$ may be pruned.

This simple example shows the value in capturing combinatorial structure within a model. Specifying this same combinatorial structure within an *Integer Programming* solver requires a *linear encoding*. A *linear encoding* refers to the process of introducing additional variables and constraints to capture the non-linear relationships between variables with only linear constraints. In the case of *alldifferent*, we can encode this structure by computing the set of

all values that could appear in the *alldifferent*:

$$D_{\text{all}} = \bigcup_{i=1}^{n} D(x_i)$$

and introducing sets of auxiliary variables for each variable x_i in the *alldifferent*. These sets of auxiliary variables are called *binarizations* of x_i and introduce new binary variables $x_{i,v}$ for each value $v \in D_{\text{all}}$. The binary variable $x_{i,v} = 1 \iff x_i = v$. Hence, as part of the *binarization* technique, the following constraints must be added to the model:

$$\sum_{v \in D_{\text{all}}} x_{i,v} = 1, \qquad \forall i \in 1 \dots n \tag{1.1}$$

$$\sum_{v \in D_{\text{all}}} v * x_{i,v} = x_i, \qquad \forall i \in 1 \dots n \tag{1.2}$$

Once a *binarization* exists for each of the variables $x_1 \dots x_n$, the *alldifferent* requirement can be enforced with the following constraints.

$$\sum_{i=1}^{n} x_{i,v} = 1, \qquad \forall v \in D_{\text{all}}$$

The *linear encoding* forces the user to think about a problem in terms of linear inequalities and auxiliary variables rather than in terms of natural high-level combinatorial structures captured by *global constraints*. Furthermore, the encodings are typically quite large – in the case of *alldifferent*, requiring $n * |D|$ auxiliary variables and $2 * n + |D|$ constraints. *Global constraints* provide both a compact formulation and a semantically rich representation of the problem which can be exploited through sophisticated *progration* algorithms and machine learning techniques.

In addition to *alldifferent*, hundreds of *global constraints* exist in the literature [11]. Many of these *global constraints* were created for particular problems, but dozens are commonly used and implemented across many CP Solvers. Below are a few of the most common *global*

constraints.

element constraint $z = Y[x]$ (see [63]), where z and x are variables and Y is an array of variables. This constraint allows a variable to be used as an index into an array.

reification $z \iff C(x_1, \ldots, x_n)$ (see [33]), where z is a Boolean variable and C is a constraint on integer variables $x_1, \ldots, x_n$. Reification allows the feasibility of a constraint C to be bound to a Boolean variable. Many solvers restrict C to be an algebraic constraint as (until recently [33]) implementing reification required a propagator for both C and the negation of C which wasn't possible to implement in general for *global constraints.*

global cardinality constraint The GCC (see [1]) is a generalization of *alldifferent* and operates on a set of integer variables X, a set of values V and a set of intervals I in one-to-one correspondence with V. That is, every value $v \in V$ has a corresponding interval $l_v \ldots u_v$ in I. The GCC requires that for every $v \in V$, the number of variables in X assigned to v must fall in the interval $l_v \ldots u_v$.

regular The regular constraint (see [67]), *regular*(S, E), requires a fixed-length sequence of integer variables $S = \langle x_1 \ldots x_n \rangle$ to represent a string generated by the regular expression E. The regular constraint is frequently used in scheduling and rostering applications.

For a more formal and extensive discussion of common global constraints, refer to the Global Constraint Catalog [11]

1.3.2 Propagators and Local Consistency

Propagation, or *domain filtering*, is one of the central features of *constraint programming.* Each constraint in a CP model has at least one associated *propagator* or domain filtering algorithm. *Propagation* typically occurs following changes to any of the domains of the decision

variables. The idea is that the semantics of the *constraints* can be leveraged to identify and remove values within a domain that cannot participate in a *feasible solution*. Since *global constraints* capture large scale combinatorial structures of a problem, they typically offer powerful filtering algorithms, but propagators also exists for simpler algebraic and logical constraints. The ability to leverage powerful domain filtering is essential for the success of CP as problems typically have exponential search spaces, and the ability to eliminate large sections of the search tree through domain filtering is crucial for tractability.

An important property of a *propagation* algorithm is the concept of *local consistency*. *Local consistency* classifies exactly what sets of values an algorithm is capable of discarding from variable domains and, hence, provides a classification of how 'powerful' the filtering algorithm is. Among the most common definitions of *local consistency* is that of *domain consistency* (also called *generalized arc consistency* or *hyper-arc consistency*). Informally, this notion of consistency requires that for each variable x, and each value $v \in D(x)$, the assignment $x = v$, participates in some feasible solution. More formally (definitions from [18]):

Definition 5 (Domain Consistency) *A constraint C is domain consistent where vars$(C) = \{x_1, \ldots, x_n\}$, if for each variable x_i, $i \in 1 \ldots n$ and for each $v_i \in D(x_i)$, there exist values $v_j \in D(x_j)$, $j \in 1 \ldots n$, $j \neq i$ such that $\{x_1 = v_1, \ldots, x_i = v_i, \ldots, x_n = v_n\}$ is a feasible assignment on C.*

Propagators capable of *domain consistency* level filtering are ideal (such as *alldifferent* [42]), unfortunately, not all constraints have efficient algorithms capable of achieving this. Weaker forms of local consistency have also been created, such as *bound consistency* (or *interval consistency*). Informally *bound consistency* views a variables domain as an interval rather than a set of discrete values and requires that just the maximum and minimum values of a variable's domain interval participate in a solution. Formally:

Definition 6 (Bound Consistency) *A constraint C is bound consistent where vars$(C) = \{x_1, \ldots, x_n\}$, if for each variable x_i, $i \in 1 \ldots n$ and for each $v_i \in \{\min D(x_i), \max D(x_i)\}$,*

there exist values $v_j \in \min D(x_j) \ldots \max D(x_j)$, $j \in 1 \ldots n$, $j \neq i$ *such that* $\{x_1 = v_1, \ldots, x_i = v_i, \ldots, x_n = v_n\}$ *is a feasible assignment on* C.

Fast *bound consistency* algorithms exist for the large majority of commonly used constraints. In practice, it is common for propagators to provide filtering that is somewhere in between *domain consistency* and *bound consistency*. For example [71] provides a fast *bound consistency* algorithm for the *cardinality* constraint, but suggests that implementations also filter additional values when it is cheap to do so.

Development of efficient *propagators* has been a major area of research in CP from its inception. Many constraints have multiple propagation algorithms with different algorithmic complexities and filtering capabilities. Some problems benefit from fast, but weak propagation while others do better with slower and stronger propagation. Many solvers take the approach of frequently running a fast propagator and then periodically running a slower and stronger propagator.

Other notions of local consistency have been developed. For a more thorough overview, refer to [9].

1.3.3 Arc Consistency Algorithms and Fixedpoints

Propagators are capable of providing *local consistency* at the level of individual constraints, but higher-level *algorithms* are still needed to find *local consistency* for the larger system of constraints that define the solution space of the problem. In particular, it is not sufficient to simply run each *propagator* once at each node of the search tree as filtering generated by later *propagators* may result in domains no longer being consistent with respect to earlier propagators. That is, additional filtering may be achieved by rerunning some of the propagation algorithms. Hence, a CP solver must implement an *arc consistency* algorithm (or *kernel algorithm*) which coordinates the execution of *propagators* until a *fixedpoint* is reached. A *fixedpoint*, represents a state in which all domains are locally consistent with

respect to all the *propagators* in the constraint system (i.e., no additional filtering can be realized by rerunning any of the propagators).

The simplest of these algorithms, known as *AC-1*, simply reruns every *propagator* any time any domain in the problem changes. This algorithm only terminates when a full pass through all the *propagation algorithms* has been made without any domain changes. Hence, the algorithm is easy to implement, but incredibly inefficient and never used in practice.

A slightly more sophisticated algorithm is *AC-3* [52]. *AC-3* provides a good balance of performance and simplicity and is the most commonly used algorithm in practice. The idea is that instead of rerunning all propagators every time a domain changes, the algorithm only reruns those propagators that may have been effected by the domain change. This is often implemented using an *observer pattern* where a propagator for a constraint, C, subscribes to events on the domains of $vars(C)$. When a propagator receives notification that a relevant domain has changes, it may immediately rerun the filtering algorithm or place itself in a queue to be run at some point before the next *fixedpoint* is reached.

Another algorithm that is useful for some propagators is *AC-5* (refer to [91]). *AC-5* is typically implemented using an *observer pattern* similar to *AC-3* but is capable of providing events that track which values are removed from a domain. Filtering algorithms for some propagators (the *element* constraint for example) can be much more efficient when notified with the exact values removed from a domain rather than simply trying to infer filtering from the entire updated domain itself. *AC-5*, however, is not cheap as extra work and messaging is required. In practice, *AC-5* is often combined with *AC-3* such that *AC-5* level messages are only provided for propagators that substantially benefit from the enhanced notification, while the rest of the propagators receive standard *AC-3* messages.

1.3.4 Search and Backtracking

A *search algorithm* defines how the search space of a problem is explored. The search algorithm can be though of as defining a *tree* composed of *search nodes* (or subproblems)

defined by *branching decisions*. Every search begins at a *root node*, denoted N_0, representing the full problem to be solved. If the problem is formulated as a model, $\mathcal{M}$, then N_0 captures the original variable domains as specified in $\mathcal{M}$ before any branching or filtering has occurred (except potentially some preprocessing).

Definition 7 (Search Node) *Given a CSP $\mathcal{M} = \langle X, D, C \rangle$ with $k = |X|$, a search node $N = \{d_1 \subseteq D(x_1), d_2 \subseteq D(x_2), \ldots, d_k \subseteq D(x_k)\}$.*

Definition 8 (Branching Decision) *A branching decision on a variable $x_i \in X$ at a search node N defines a partition $p_{i,1}, p_{i,2}$ of the domain $d_i \in N$. The partition of d_i induces a partition on the problem N into two subproblems (or subnodes):*

- *N_1, defined by substituting $p_{i,1}$ for d_i in N.*
- *N_2, defined by substituting $p_{i,2}$ for d_i in N.*

Any search algorithm then begins at a start node (typically the root node) and is defined by two components:

variable-ordering heuristic A procedure for selecting which variable to branch on next.

value-ordering heuristic A procedure for determining what value to assign to the selected variable.

These heuristics may be dynamic, changing as the search proceeds. Figure 1.4 provides a flow chart of a typical CP search procedure at a high-level. The procedure begins by adding a root node N_0 to the list of open nodes $\mathcal{L}$, setting the current upper bound $\bar{z}$ to infinity and the incumbent solution s^* to $\emptyset$. The search proceeds by removing a node N_i to explore and continues until the list of open nodes is empty. The node N_i is first restored, meaning that variables domains are reset to the state they were in when node N_i was captured (as well as related data structures and search parameters). The variable and value ordering heuristic are then used to branch on a variable (say the assignment $x = v$ is made). At this

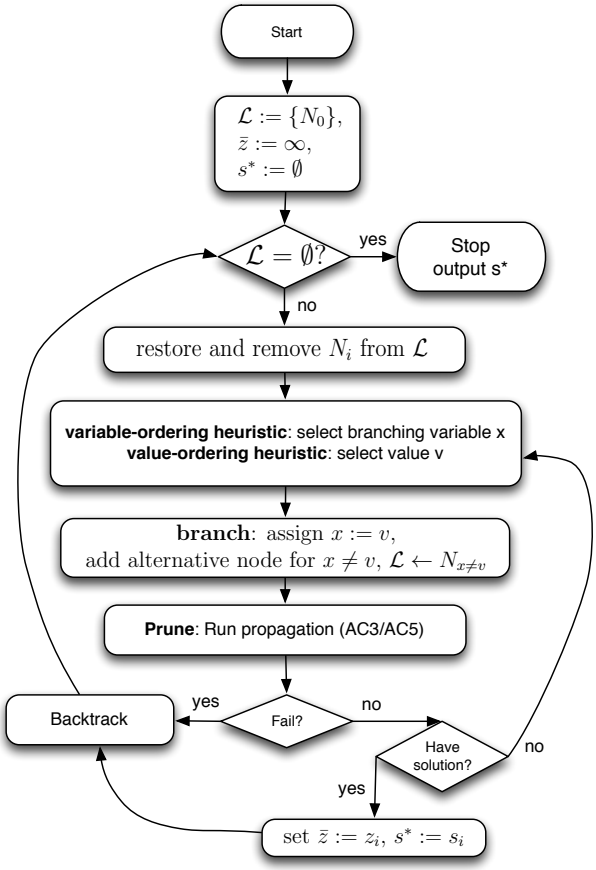

Figure 1.4: High-level overview of CP search

point, an alternative node with a new branching constraint is added to $\mathcal{L}$ (prohibiting x from taking on the value v in this case) for later exploration and propagation is triggered. Note that the the upper bound $\bar{z}$ also provides pruning by requiring that the objective function satisfy the constraint $f(x) < \bar{z}$. If the propagation process results in a failure, then the search backtracks to an earlier node in the search tree by restoring some node from $\mathcal{L}$. If, on the other hand, the propagation, successfully reaches a fixedpoint, the process of assigning variables can continue. Once all variables have been assigned, a feasible solution has been found and the upper bound ($\bar{z}$) and incumbent (s^*) can be updated. Note that the solver always imposes $f(x) < \bar{z}$ during the search procedure and, hence, any time a solution is found, it necessarily represents an improvement over the previous incumbent.

Effective search is crucial as finding high-quality solutions quickly allows the solver to use the objective value of the incumbent solution to prune the remaining search space. Furthermore, some variables often influence the objective function more dramatically than others and the search can typically do better if these variables are branched on first. In general, designing an effective search can be challenging and time consuming. Fortunately many good 'blackbox' searches exist and can be tried before a user resorts to manually writing a search (for a description of some blackbox searches, see section 5.2.3).

1.4 Integer Programming

Integer Programming (IP, see textbooks [61], [24]) is the best-studied and most widely used *discrete optimization* technology. IP models take the following form:

$$\mathcal{M} = \min c^T x$$
$$\text{subject to}$$
$$\left\{ Ax \leq b \right.$$

Here x is a vector of discrete decision variables, c is a vector of objective coefficients, and $Ax \leq b$ defines a system if linear inequalities. If x is allowed to be a mix of integer and continuous variables, then the problem is described as a *Mixed-Integer Programming problem* or MIP. Problems with only continuous variables are *Linear Programming* problems and can be solved efficiently using either the *simplex method* (average-case polynomial time) or the *interior-point method* (worst-case polynomial time). Note that IP and MIP models are a more restricted form of the general *optimization models* provided in the introduction as both the *objective* and *constraints* are linear.

The procedure used by modern IP and MIP solvers (for example Gurobi [43]) is a linear programming branch-and-bound algorithm. Since the integrality requirement on the variable domains cannot be handled directly, the algorithm proceeds by iteratively solving a series

of *LP relaxations* of the original model. An *LP relaxation* of an IP model simply replaces discrete variables with continuous ones transforming the problem from a *discrete optimization* model to a *linear programming* one. A simplified sketch of the algorithm is shown in Figure 1.5 and is an adaptation of the branch-and-bound method described in [24]. Although this figure provides a high-level conception of how a MIP is solved, modern high-performance MIP solvers rely on many subtle and technical additions to this algorithm including sophisticated numerical error handling, complex branching heuristics and optimizing for sparse matrices.

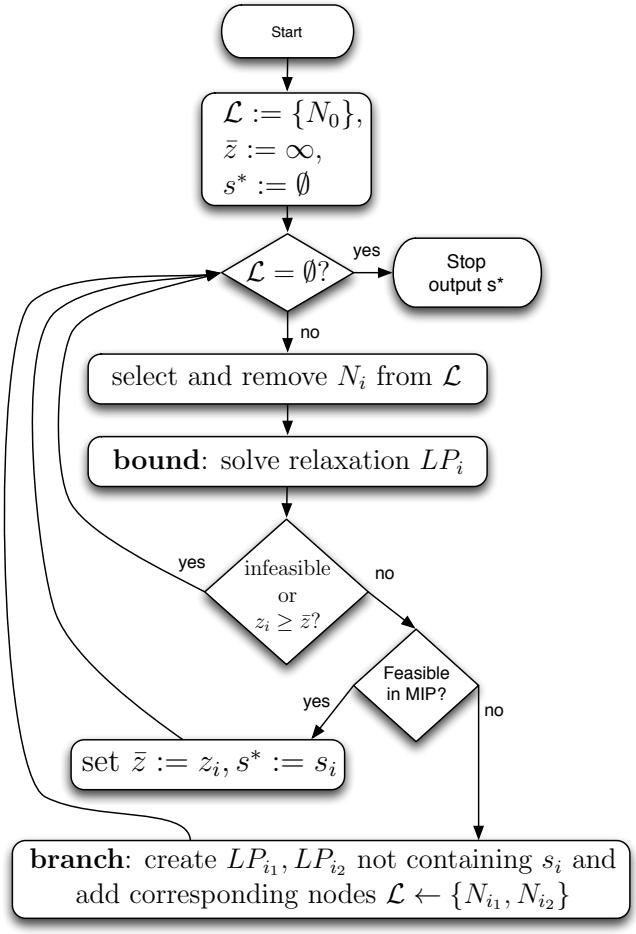

Figure 1.5: High-level overview of MIP branch-and-bound search

The search procedure in Figure 1.5 begins in a similar manner to the CP search described earlier. An open node list $\mathcal{L}$ is initialized with a root node N_0, the incumbent s^* is set to $\emptyset$ and

the upper bound $\bar{z}$ is set to ∞. The search proceeds until $\mathcal{L}$ is empty, removing a node from the list and solving the *LP relaxation* (typically by the simplex method). If the relaxation is infeasible or the solution is worse than the incumbent solution, then the search returns to the list and tries another node. If the solution to the relaxation is an improvement over the incumbent, then the search must determine if the solution also satisfies the original IP (i.e., all variables are assigned integral values). If this is the case, then the solution becomes the new incumbent and the bound is updated. If the solution does not satisfy the original IP, then the relaxation is broken into subproblems where the current solution to the relaxation becomes infeasible, but no integral solutions to the problem are lost. This can be done by choosing a variable assigned to a non-integral value, say $x = v$, and creating a subproblem with the additional constraint $x \geq \lceil v \rceil$ and another subproblem with the constraint $x \leq \lfloor v \rfloor$.

1.4.1 Linear Programming Duality

Duality refers to the fact that every linear programming problem (called a *primal*) has a closely related problem called its *dual* problem. Which program is called the *primal* and which is the *dual* is a matter of semantics because the *dual* of the *dual* is the *primal*. *Primal* and *dual* problems are formulated as follows:

$$\mathcal{M}_{primal} = \min c^T x \qquad\qquad \mathcal{M}_{dual} = \max b^T y$$

$$\text{subject to} \qquad\qquad \text{subject to}$$

$$\left\{ Ax \leq b \right. \qquad\qquad \left\{ yA \geq c \right.$$

Linear Programming Duality is covered in detail in many textbooks including [61], which the description here is adapted from. One of the useful features of the *dual* (assuming the *primal* is a minimization problem) is that any solution to the dual provides a lower bound on the *primal* objective. Furthermore, this lower bound can be made arbitrarily tight. This is formalized in the theorems of *weak duality* and *strong duality*. For proofs of these theorems

refer to [61].

Theorem 1.4.1 (Weak Duality) *If $\bar{x}$ is a feasible solution to $\mathcal{M}_{primal}$ and $\bar{y}$ is a feasible solution to $\mathcal{M}_{dual}$, then $b^T\bar{y} \leq c^T\bar{x}$.*

Theorem 1.4.2 (Strong Duality) *If x^* is an optimal feasible solution to $\mathcal{M}_{primal}$ and y^* is an optimal feasible solution to $\mathcal{M}_{dual}$ and either b^Ty^* or c^Tx^* are finite, then $b^Ty^* = c^Tx^*$.*

Duality is a central concept in LP and has many uses, some of which are relevant to this thesis, including:

1. Finding *reduced cost* in column generation (refer to section 3.3).
2. Generating cuts in classic *Bender's Decomposition* [13].
3. Duality is generalized in the *Logic-Based Benders* technique, introducing the notion of a *logical dual* (see section 3.4).
4. A related notion of duality, called the *lagrangian dual* is iteratively solved in *lagrangian relaxation* (see section 3.6).

1.4.2 Modeling in Integer Programming

Integer Programming is built on the strategy of leveraging efficient *Linear Programming* algorithms for use with *discrete optimization problems*. As a technology, this has worked out remarkably well for many decades. Integer Programming is still the most popular, and for many problems, the most effective *technology*. An unfortunate drawback of the IP strategy, however, is that users have to *model* problems as if they are LPs. That is, *Integer Programs* may only be expressed using linear inequalities. Since combinatorial optimization problems typically have complex non-linear structure, users must leverage a myriad of *encoding* and *approximation* techniques to express their problems as an IP. These techniques require adding large numbers of auxiliary variables and constraints resulting in ballooning model sizes.

Choosing an encoding that is both correct and efficient is an art form that can take years of experience to develop (see textbook [61]). Below are some of the encoding techniques commonly employed in IP:

Binary Decision Variables Binary variables play a central role in a large number of IP problems. Such variables often represent a choice, such as locating a warehouse at a particular location or not (1 or 0). If we have a warehouse that can be located at one of ten locations, this can be captured using binary variables as follows:

$$\sum_{i=1}^{10} x_i = 1$$

Here, the x_i's are Boolean variables indicating if the warehouse is located in any of the ten positions under consideration. Since the warehouse may ultimately only be placed in one of these locations, the sum of these Boolean variables must be exactly 1.

Disjunction Disjunctions appear frequently in *combinatorial optimization* and one of the challenges of IP *encoding* is finding creative ways of modeling the problem to avoid inefficient encodings of disjunctions. A general, and often necessary, way of encoding a disjunction relies on the so-called 'Big-M' technique. The idea is to make use of a very large constant M to allow the solver to switch particular constraints on or off. Consider the following *encoding* of the logical constraint, $x_1 > x_2 \vee x_1 < x_2 - 3$:

$$x_1 + z * M > x_2 \tag{1.3}$$

$$x_1 - (1 - z) * M < x_2 - 3 \tag{1.4}$$

A binary variable z has been introduced to allow one of the two constraints to be ignored. In particular, when $z = 0$, the top constraint is equivalent to $x_1 > x_2$, while the bottom constraint is trivially satisfied as a very large number M is subtracted from x_1 making it much smaller than $x_2 - 3$. The opposite effect is achieved when

$z = 1$. Hence, by adding an auxiliary Boolean variable and two 'Big-M' constraints, the disjunction has been encoded. Note that since we know the initial domains of x_1 and x_2, it is possible to calculate how large of an M is needed to make this work. Keeping M small is important as the use of very large values of M often yields poor bounds on the objective and overall poor solver performance.

reification As with CP, reification allows the feasibility of a constraint C to be bound to a Boolean variable. In IP, this can be encoded using the 'Big-M' method. Suppose that we have a Boolean variable z and would like it to be 1 *iff* the constraint $ax \leq b$ is satisfied:

$$ax - (1 - z) * M \leq b \tag{1.5}$$

$$ax + z * M > b \tag{1.6}$$

This example is similar to the *disjunction* case, where the value of z will 'switch off' one of the two constraints. Notice that the second constraint is needed to force z to be 1 when the constraint, $ax \leq b$, is satisfied.

1.4.3 Example: IP formulation of Warehouse Location Problem

The Warehouse Location Problem (WLP) was first introduced in section 1.2.3. The formulation provided previously made use *element constraints* and *reified expressions* not directly applicable to IP. To get an idea of how modeling in *Integer Programming* works, an IP model for WLP is provided, using several of the *encoding techniques discussed above.*

$$\mathcal{M} = \min \sum_{s=1}^{n} cost_s + \sum_{w=1}^{m} fixed_w * open_w$$

subject to

$$\begin{cases} \sum_{w=1}^{m} support_{s,w} = 1, & \forall s \in 1 \dots n \\ cost_s = \sum_{w=1}^{m} support_{s,w} * C_{s,w}, & \forall s \in 1 \dots n \\ open_w * n \geq \sum_{s=1}^{n} support_{s,w}, & \forall w \in 1 \dots m \\ \sum_{s=1}^{n} support_{s,w} \leq capacity_w, & \forall w \in 1 \dots m \end{cases}$$

Here, we again have $1 \dots n$ stores and $1 \dots m$ warehouses and the corresponding *cost* and *open* variable vectors. The *support* vector from the earlier model has been replaced by an $n \times m$ Boolean variable matrix. The semantics of this matrix is that $support_{s,w} = 1$ *iff* store s is supplied by warehouse w. Both the *open* vector and the *support* matrix provide an example of using Boolean variables to model different potential decisions. Furthermore, the transformation of the *support* vector in 1.2.3 into the matrix in the example can be seen as an example of *binarizing* the variables in the vector. Furthermore, the constraint:

$$open_w * n \geq \sum_{s=1}^{n} support_{s,w}, \quad \forall w \in 1 \dots m$$

Is really just a reification of, $open_w \iff \sum_{s=1}^{n} support_{s,w} > 0$, with n playing the role of a tight 'Big-M' constant. Notice in the discussion of *encoding reification* above, two constraints were required. In the case of WLP, only the single constraint is needed because $open_w$ appears in the objective resulting in the solver assigning $open_w = 0$ whenever possible, obviating the need for a second constraint enforcing the *iff* condition.

1.5 Local Search

Local Search (see textbook [2]) encompasses a broad literature of *improvement algorithms* which begin with an incumbent assignment, called the *solution* (though possibly infeasible), and iteratively explore a neighborhood around this solution trying to make improvements.

Unlike a typical search algorithm used in CP or IP, local search is an incomplete technique and is best visualized as a walk on a directed graph as opposed to a traversal of a search tree. For a CSP, local search algorithms try to improve the solution by reducing constraint violations until a feasible solution is found. In the case of a COP, local search tries to improve the objective function, where the objective may include terms representing constraint violations. Since local search techniques are incomplete, there is no natural stopping condition for the search. Users typically provide stopping conditions such as a limit on the number of trials, a time limit, a no improvement limit or some condition on the solutions.

Every local search algorithm is built on three basic operations:

Neighborhood Operation (N) defines a set of neighboring solutions given the current solution.

Legal Operation (L) partitions a set of neighbors (identified by N) into a *legal* set, indicating solutions that are allowed moves, and a *forbidden* set indicating invalid moves.

Selection Operation (S) Selects a neighbor from among the legal moves produced by L and decides whether or not to move to that neighbor or stay at the current solution.

Figure 1.6 provides a diagram of a generic local search procedure (adapted from [92]). The search starts by generating an initial solution, s. While the stopping condition is not met, s is updated using the N, L and S operators moving the search on a walk through the search space. Each time a satisfiable solution is found that improves on the incumbent, the incumbent is updated, $s^* := s$.

The N, L and S operators can be arbitrarily sophisticated and often leverage data structures for maintaining information on visited nodes or neighborhoods. One of the best know local search heuristics, *TabuSearch* ([92]), maintains short and long term memory data structures recording recently visited nodes and also recording high quality solutions to return to periodically.

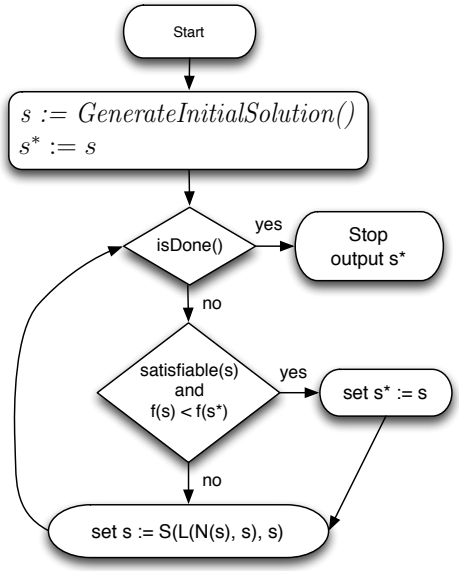

Figure 1.6: High-level overview of generic local search algorithm

Local search strategies can be leveraged on top of existing technologies such as CP and IP or they may be used from independent local search technologies such as *Constraint-Based Local Search* (CBLS) [92]. CBLS is built from the ground up using many of the same constraints and modeling techniques as CP, but provides a number of features including *invariants*, *differentiable objects*, blackbox search heuristics and a sophisticated syntax and standard library for building complex local search algorithms (refer to textbook [92]).

1.5.1 Large-Neighborhood Search

Large-Neighborhood Search (LNS) is a popular local search technique first proposed by Shaw in [82] and typically implemented on top of a full CP or IP solver. The process is based on a continual relaxation and re-optimization of parts of the problem using a complete CP or IP search. Abstractly, LNS can be viewed as having two generic operations:

Fix defines a subset of problem variables $X_R \subset X$ to be relaxed.

Freeze which assigns relaxed variables in X_R to fixed values.

The result of fixing and freezing some subset of the decision variables is to create a relaxation of the original problem in which the solver only needs to optimize over variables in $X - X_R$. Hence, the solution returned by the solver is not globally optimal, but only optimal with respect to the neighborhood defined by X_R. Once the solver has optimized the relaxed problem for the neighborhood X_R, the neighborhood is modified and then re-optimized. The neighborhood can be modified by changing the fixed values in X_R, by removing some relaxed variables in X_R and reintroducing them into the problem, or some combination of both. LNS is an abstract technique and the details of *fix* and *freeze* operations will be specific to each problem. A detailed discussion of an LNS algorithm for the IMRT problem can be found in Chapter 7.

Chapter 2

Thesis Statement

2.1 Motivation

Solving combinatorial optimization problems is a very technical and fragile process, typically requiring years of experience and advanced degrees to be able to do well. This is due, in part, to the nature of combinatorial problems themselves. Problems in this category are typically *NP-hard* with exponential sized search spaces, meaning they are inherently difficult to solve. The structure of these problems are also idiosyncratic and, hence, strategies that are effective in finding good solutions for one particular problem may prove to be useless on the next. Even worse, strategies that work on one instance of the a problem, may not work when the instance is changed slightly. Given these realities, combinatorial optimization problems are destined to remain generally difficult to deal with.

On the other hand, part of the difficulty involved in combinatorial optimization stems from the fact that the field still lacks the kinds of tools and abstractions that make common techniques from the literature readily accessible. As described in the introduction, distinct technologies exist to model and solve these problems and each technology bring with it different strengths and weaknesses. Constraint Programming relies on aggressively pruning the search space by capturing the large combinatorial structure of the problem and exploiting

it through logical inference. Integer Programming exploits fast and efficient algorithms that exist for LP by iteratively solving linear relaxations of the problem until a solution is found. Other technologies such as SAT and SMT also bring unique algorithmic approaches to combinatorial optimization and can be very effective on certain categories of problems.

Beyond distinct algorithmic approaches, each technology also has modeling requirements and idioms which must be mastered before the technology can be used effectively. For example, a user who has spent years modeling with MIP may struggle to effectively model a problem in CP. MIP modelers think in terms of inequalities, 0-1 variables, and complex summations rather than global combinatorial structure. Hence, when they model a problem in CP in the manner they are accustomed to in MIP, they are likely to be hit with very poor results. Similarly, a CP user may struggle to even correctly formulate their problem as a MIP or they may formulate it in a manner that yields very weak bounds. Indeed, a brief look at any Integer Programming textbook will confirm the extreme importance of effective modeling (for example [61]).

Even when a problem is modeled well using an appropriate technology, it is often the case that it is still unsolvable. This has lead to a very large literature of 'hybrid' approaches. The word 'hybrid' here is used to describe not only the case when multiple technologies are employed, but also includes any of the sophisticated techniques that go beyond the typical model and run approach. This includes techniques which split a problem up and solve it in stages, techniques which iteratively solve relaxations or incomplete problem formulations and problems that leverage LNS or local search techniques alongside complete searches. Examples include *lagrangian relaxation*, *column generation* and *bender's decomposition* (refer to textbook [24] for an overview of all three examples).

Despite the fact that many of these hybrid techniques have been known and used with great success for decades, they still are implemented from scratch on a case-by-case basis. This has been very problematic for the field, having several major drawbacks:

Wasted time Many of these hybrids require significant amounts of coding and even more

debugging every time they are implemented.

Powerful hybrids are under-utilized Due to the expertise involved in properly coding these techniques as well as the substantial investment of time and effort, many potentially useful hybrids are never applied to intractable problems.

Mistakes and results which can't be duplicated The fact that these hybrids are often complicated and subtle to implement along with the reality that users are trying to implement them based on dense and sometimes incomplete academic papers results in incorrect implementation (recent example [21]). Incorrect implementation leads to incorrect or misleading results which lead other researchers down an unfruitful path or perhaps to discard a promising idea.

2.2 Thesis Statement

The purpose of this thesis is to introduce new tools and theoretical abstractions that make it easy to rapidly reformulate a model for use with different solver technologies and hybrid techniques. The mechanism for doing this is through a new 'algebra' of *abstract models and operators*. *Abstract models* represent a high-level, technology neutral problem description. Several types of *operators* may be used to manipulate an *abstract model*:

transformation operator An abstract model is mapped to another abstract model. The new model may be semantically equivalent (retains the same solution set), a relaxation (expands the solution set) or a tightening (reduces the solution set).

concretization operator An abstract model is transformed into a concrete, technology-specific solver.

composition operator Provides automation for leveraging existing solvers within a particular hybrid framework. The result is another solver which may be further composed into increasingly more sophisticated hybrids.

These concepts will be greatly expanded upon and formalized later, but the remainder of this chapter will be devoted to sketching the high-level vision of this work.

2.2.1 Vision

The process of generating a hybrid from a high-level problem description is not simply an engineering exercise. Solvers and hybrids have very particular semantics and creating generic operators that create meaningful compositions requires the development of some new theoretical tools. Classic hybrid techniques, such as *lagrangian relaxation* have always been used within a linear programming context. Creating a generic *lagrangian relaxation* operator requires more than simply plugging a model into a template, but rather, requires a re-imagining and reinterpretation of what the technique actually is at a high-level.

Example: To give a sense of the vision, consider the *warehouse-location problem* (WLP, introduced in section 1.2.3). The problem may be described at a high level as an abstract model:

$$\mathcal{M} = \min \sum_{s \in Stores} cost_s + \sum_{w \in Warehouses} fixed_w * open_w$$

subject to

$$\begin{cases} cost_s = C_{s,support_s}, & \forall s \in Stores \\ open_{support_s} = 1, & \forall s \in Stores \\ \sum_{s \in Stores} (support_s = w) \leq capacity_w, & \forall w \in Warehouses \end{cases}$$

Suppose a user wants to run this problem as a MIP, then the high-level model is first transformed into a linear formulation using a linear operator and then *concretized* into a final MIP solver that may be executed (see figure 2.1)

If running the MIP WLP proves to be unsuccessful, the user may decide they'd like to run both a MIP and CP formulation of the problem in parallel and have the solver communicate bounds to speed up the search. Typically, creating such a hybrid would be complex, requiring the user to set up multiple threads, a thread-safe communication channel, and transcoding of solutions (for a discussion see section 3.2). The aim of this thesis is to reduce the generation

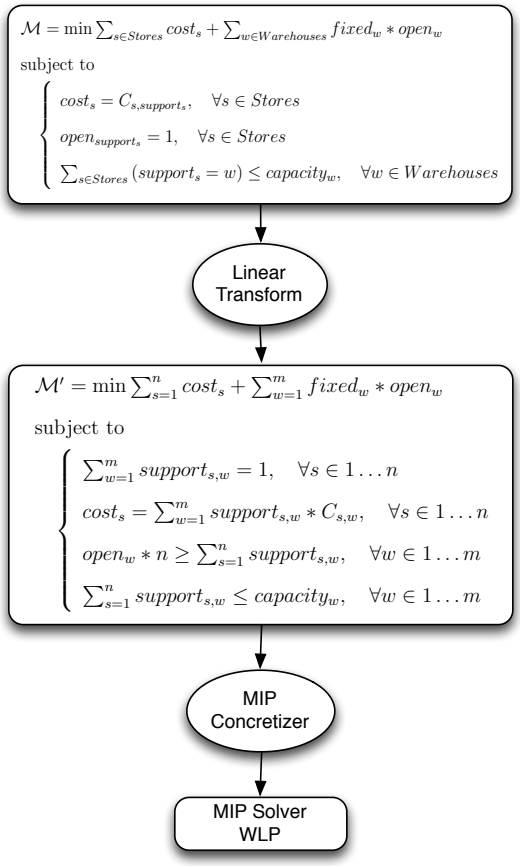

Figure 2.1: High-level model to a concrete MIP Solver

of this *parallel hybrid* into a single line of code that simply chains operators together and applies them to an abstract model. Hence, Figure 2.1 would become Figure 2.2.

In this case we start with an abstract model as before, but the same model is used to produce two different solvers, a MIP solver on the left and a CP solver on the right. The two solver are then passed into a 'parallel composition' operator which produces a final parallel hybrid solver. We could imagine further composing this parallel solver with other solvers. With the approach that will be presented in the coming chapters, this entire process can be captured in just several lines of code.

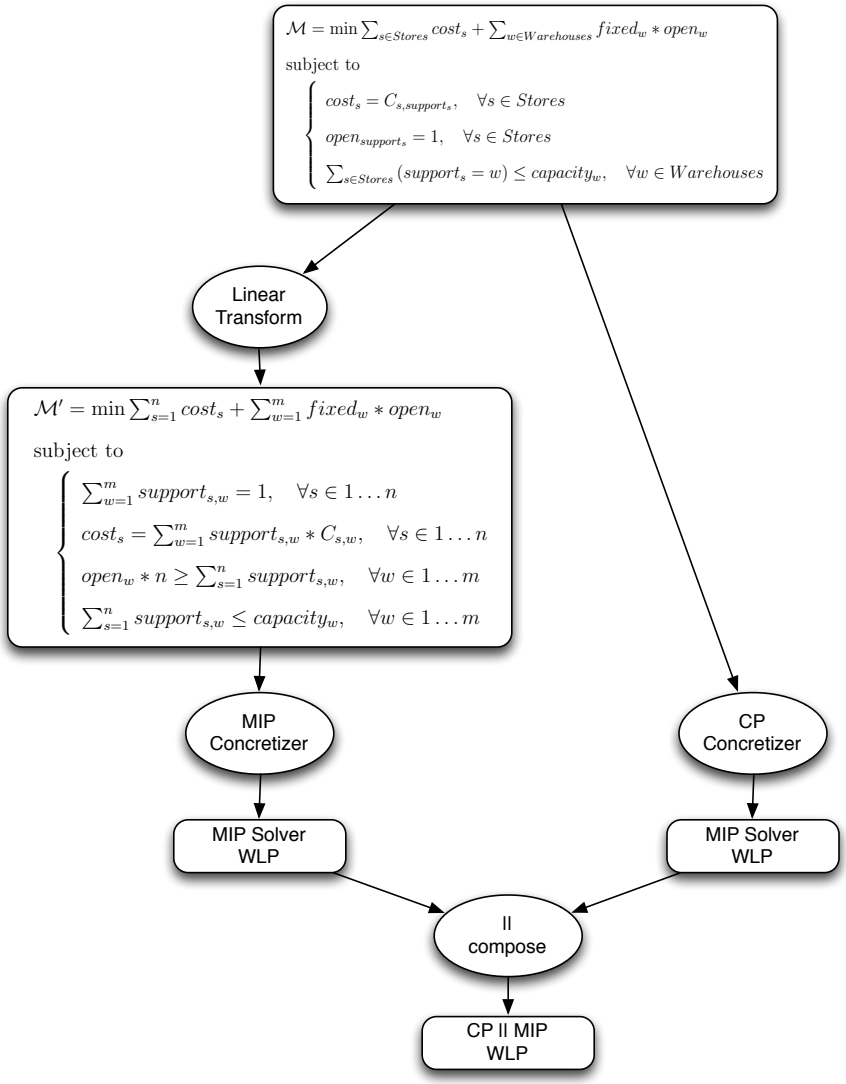

Figure 2.2: High-level model to a concrete MIP Solver

2.2.2 Semantics

The vision presented thus far is already compelling, but to make such generic operations truly useful requires capturing problem and solver semantics. In particular, if the user attempted to apply the 'parallel compose' operator on a MIP solver for the WLP with a CP solver for the *assignment problem* (section 1.2.1), the resulting solver would be ill defined, likely producing wrong solutions for both problems. An operator should be able to detect such a semantic error. Furthermore, the WLP is formulated as a *minimization problem*. The CP

solver will produce a sequence of solutions which converge on the optimal. The objective value of this sequence of solutions will yield a corresponding sequence of upper bounds on the objective. If, on the other hand, we solved an LP relaxation of the problem, we could produce lower bounds on the objective. These are properties inherent to the solver technology being used and can be captured as *metadata* that may be used in aiding in the composition of parallel solvers and detecting when an invalid composition is made. Such *metadata* may also be used to automate the composition process itself as the operator knows exactly what kinds of bounds each solver is capable of producing and what bounds each solver can consume.

Another form of *metadata* is produced by the *model transformations* themselves as they are applied to *abstract models*. In particular, notice in our example that models $\mathcal{M}$ and $\mathcal{M}'$ are closely related models. In fact, they are semantically equivalent as $\mathcal{M}'$ is simply a linear reformulation of $\mathcal{M}$. Model relationships imposed by the transformation operators may be tracked and inspected by other operators to enforce the semantic integrity of the operations as they are applied.

2.2.3 Contributions

The contributions of this thesis which will be presented in detail in the coming chapters are the following:

1. A more detailed discussion and examples of *abstract models* and *operators* and hybrid composition introduced through a custom scripting-language running on top of COMET, first published in [38].

2. A new formal framework for realizing the vision put forward in this chapter. This includes new concepts such as *model signatures*, *runnables* and *model combinators* as well as some experimental results suggesting that the proposed framework provides efficiency similar to hand-written code. This work was first published in [39].

3. An example of a new LNS approach for Intensity-Modulated Radiation Therapy and

a description of how this custom search can be easily implemented with a generic LNS operator (First published in [25]).

4. A proposal for moving *portfolio solver* techniques to a coopertative rather than competitive approach by leveraging *parallel composition operators*. Parallel solvers combining CP and MIP on scheduling problems are presented for the first time. (Published [40])

5. A generalization of lagrangian relaxation techniques and a combinator allowing generic lagrangian relaxation to run on top of MIP, CP and LNS solvers (published [37]).

Chapter 3

Patterns of Composition

This chapter outlines some of the hybrids and composition patterns that will be automated in the remainder of the thesis. This list is far from exhaustive, but captures a number of well known and successful hybrid techniques from literature that can be difficult or cumbersome to implement from scratch.

3.1 Sequential Hybrid

A commonly used composition pattern involves breaking the process of solving a problem into multiple steps. This approach is often taken to find a good bound on the objective before a complete search is undertaken. For example, a user may run an LNS or Local Search heuristic for a set amount of time to find a good upper bound on a minimization problem. After the time limit on the local search technique expires the bound is passed to a complete search which can make use of it to more aggressively prune the search space. Similarly, a user may choose to run an LP relaxation of a problem to find a lower bound to be leveraged later within a complete CP search (CP is feasibility driven, unlike IP which relies heavily on bounds during optimization).

In terms of implementation, *sequential hybrids* are a simple composition patterns, involving writing two independent solvers and a bit of code to pass information between them

(see Figure 3.1). Although the composition itself is rather simple, there is still quite a bit to gain here from automation. In particular, the models being run are often related to one another. That is, the models may have equivalent solution sets or one of the models may be a relaxation of the other. In this case, it would be ideal to model the problem once at a high-level and then make use of model operators to reformulate the problem. Furthermore, if the models are different formulations of the same problem, leveraging automation would allow easy transcoding of solutions between the two models.

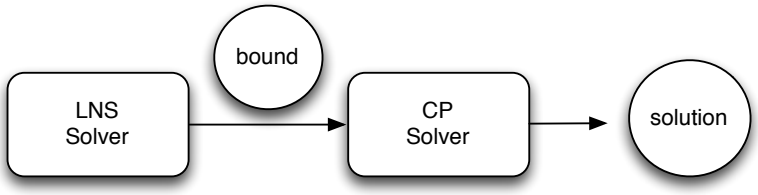

Figure 3.1: Example sequential composition pattern

3.2 Parallel Hybrid

Parallel hybrids are simple to understand conceptually, but often difficult to implement correctly in practice. The motivations behind this composition pattern are often similar to the case of sequential hybrid, but rather than running the solvers independently of one another in sequence, the two solvers run concurrently sharing bounds on the objective bi-directionally and in real time. Furthermore, the technique has the potential to be more powerful than sequential composition as the bounds sharing and pruning benefits go both ways sometimes resulting in rapid and unpredictable advances in the search.

Difficulty in implementing a parallel hybrid has several causes. If different technologies are involved, writing multiple encodings of the same problem using different solver backends can be time consuming (just as in the sequential case). Running in parallel requires setting up several threads and coordinating real time communication across asynchronous thread

boundaries as new bounds become available (see Figure 3.2). As bounds are received by a solver they must be stored until the search is in a state where the bound can safely be injected (typically when starting a new node in the case of MIP and CP). In the case of some IP solvers, such as Gurobi, the bound is updated via the injection of an entire solution. This requires the transcoding of the solution from one model formulation (CP or LS) to a radically different one (IP), a cumbersome process in its own right. Additionally, the startup and shutdown of solvers also requires careful implementation as the communication of bounds should not commence until both solvers have begun the search process. For example, an attempt by the CP solver to inject a solution into the MIP solver during the MIP's presolve phase can lead to incorrect behavior. Similarly, when one of the solvers successfully finishes, care must be taken to cancel the remaining solver and shutdown the threads cleanly. This can be challenging as a solver will typically be deep in a search tree when a cancellation request is received. The solver must handle the delicate process of cleaning up and releasing memory associated with data structures before moving execution out of the search tree and finally canceling the thread.

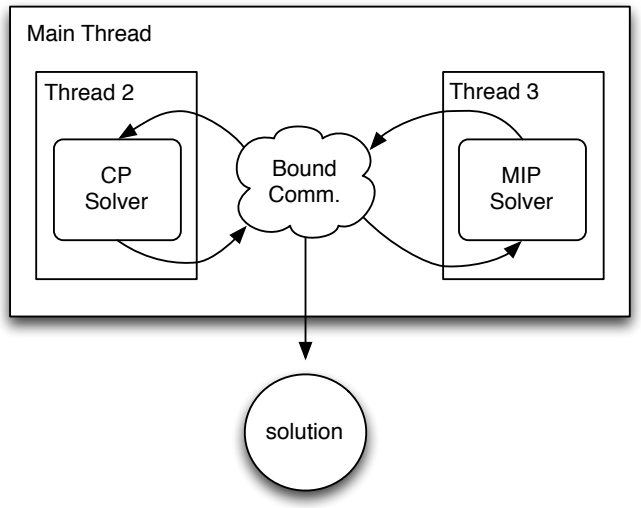

Figure 3.2: Example parallel composition pattern

3.3 Column Generation

Column Generation (also called Dantzig-Wolfe Decomposition, see textbook [24], [14]) is a classic *hybrid solver* technique created for problems with a very large (potentially exponential) number of variables, most of which will be 0 in the final solution. The basic insight is that for *LP* problems, it is possible to model the problem with a small initial subset of variables and then to add missing variables as they are needed during the optimization process. The algorithm is split among a *master problem* and a *slave problem* (also called the *subproblem*). The LP *master problem* represents the actual problem to be solved, but with a limited set of variables. The *slave problem* is an additional model built from the solution of the *master problem* and leveraged to construct a new column (and associated variable) which will allow a new solution to the *master problem* with an improved objective value. Hence, *column generation* is an iterative process which repeatedly solves the *master problem*, constructs and solves a *slave problem* and then injects the new column back into the *master*.

Historically, *column generation* was developed for *linear programs*, but in recent decades, it has become common to use *CP* within the *slave problem* making it a true multi-technology technique. Column generation works be leveraging the *linear programming dual* (refer to section 1.4.1 or a textbook such as [61]). The solution to the dual can be used compute *reduced costs* for potential new variables and associated columns. The *reduced cost* of a variable indicates how much the 'price' of that variable (i.e., its coefficient in the objective function) would have to change before the variable could take a positive value in the optimal solution. Hence, in the context of *column generation*, *reduced costs* can be used to determine which variables would be advantageous to add to the *master model* to permit an improved optimal solution. Recall the LP formulation from the *primal* and *dual* from section 1.4.1:

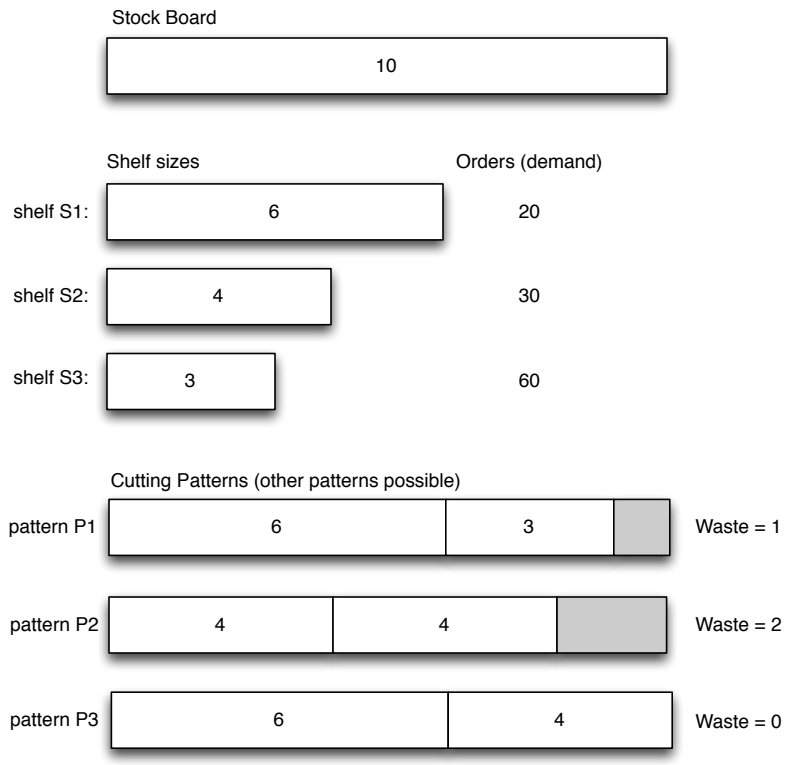

Figure 3.3: Example cutting stock problem. Orders must be cut from stock boards while minimizing waste.

$$\mathcal{M}_{primal} = \min c^T x \qquad \mathcal{M}_{dual} = \max b^T y$$

subject to $\left\{ Ax \leq b \right.$ subject to $\left\{ yA \geq c \right.$

By Dantzig's Rule, the *reduced cost* is $c - A^T y$ (refer to [14] for a full discussion of the simplex method, reduced cost variables and column generation methods). To better understand how this process works, it is helpful to consider an example.

Example: The *cutting stock* problem consists of cuttings standard sized boards into specialized pieces to satisfy orders (see Figure 3.3). The objective of the problem is to minimize the material used. Assume the following constants have been defined:

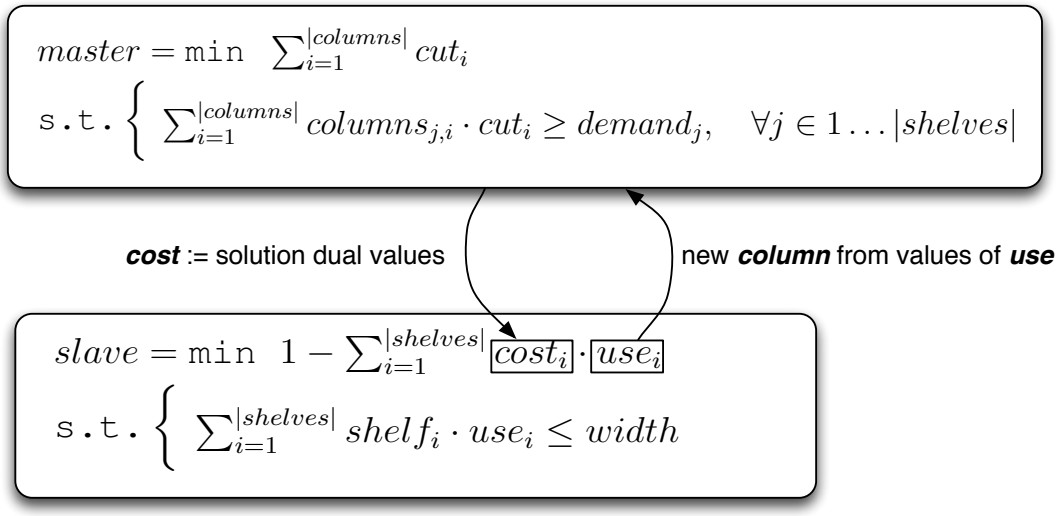

Figure 3.4: Cutting Stock problem modeled using *column generation*

width The fixed width of a stock piece of board (10 in this example).

shelf An array of specialized lengths to be cut from stock boards ($[6, 4, 3]$ in this example, corresponding to the lengths of S1, S2 and S3).

demand An array indicating the number of orders for each of the shelfs ($[20, 30, 60]$ in this example).

columns The columns represent the *patterns* used to cut a stock board. Figure 3.3 shows three initial patterns, $P1 = [1, 0, 1], P2 = [0, 2, 0], P3 = [1, 1, 0]$. The pattern captures how many of each shelf ([#S1, #S2, #S3]) to cut from a stock board.

Solving *cutting stock* using *column generation* requires formulating *master* and *slave* models. Figure 3.3 shows a high-level description of these models. The *master* problem is an LP model with a vector of integer variables called *cut*. This vector represents the number of stock boards that should be cut into each of the available patterns. The objective is to minimize the number of total stock boards to be cut. The constraints state that the number of shelves of each length that are produced must meet their respective demands.

The *slave problem* produces a new pattern (column) with the most *negative reduced cost*. Notice that the objective function is equal to the reduced cost formula (where c = 1 because all patterns have equal weight). Since the *slave* problem minimizes the reduced cost, it finds the most advantageous new pattern while adhering to the constraint that the new pattern must fit within the length of a stock board. If a solution to the *slave problem* with *negative reduced cost* doesn't exist, then the current incumbent solution for the *master problem* is optimal. If the *slave* problem is successful at generating a *reduced cost* pattern, then this new pattern is injected into *master's* matrix of linear constraints as a new column and a new variables is introduced corresponding to this new column.

In this example, solving the master problem with the initial three patterns yields an optimal solution of $P1 = 60, P2 = 15, P3 = 0$, using a total of 75 stock boards. The *cost* vector coming from the *dual* of the *master* is $[0, 0.5, 1.0]$. Solving the *slave* problem (with the *cost* vector from the master) yields a new reduced cost pattern, $P4 = [0, 0, 3]$. Injecting this pattern into the *master* problem as a new column, we can repeat the process. Resolving the *master* yields a new optimal solution $P1 = 0, P2 = 5, P3 = 20, P4 = 20$, using 45 stock boards. The *slave* can now be resolved with a new *cost* vector of $[0.5, 0.5, 0.333]$. This time, the *slave* still produces a negative reduced cost pattern, $P5 = [0, 1, 2]$. Using this pattern, the *master* is able to find a solution $P1 = 0, P2 = 0, P3 = 20, P4 = 13.33, P5 = 10$ using 44 stock boards. At this point, the *slave* cannot find a reduced cost solution, indicating that the most recent solution to the *master* is optimal.

3.4 Logic-Based Benders

Logic-Based Benders [45] is another iterative technique which is a generalization of classic Bender's Decomposition. The idea is to partition a large problem into two smaller problems called the *master* and *slave* problems. The idea is that the *master* produces a solution to part of the original problem and then the *slave* tries to extend the solution from the *master*

into a solution for the whole problem. If the *slave* cannot extend the solution of the *master* (because it is infeasible) then a *cut* is generated and added back into the *master* problem to eliminate the previous solution. Ideally, the cut is not simply a *no-good* and should bound the objective of the *master* resulting in the elimination of large portions of the search space. If the *slave* is feasible, then the procedure is complete and the solutions to the *master* is optimal. Note that a drawback of this technique is that no feasible solutions to the original problem are generated until the procedure terminates with the optimum.

Logic-Based Benders demonstrates that the concept of the 'dual' can be generalized to that of a 'logical dual.' The logical dual is produced by optimizing any subproblem which gives a lower bound on the *master objective* and prohibits the previous master solution. No general procedure exists for formulating a subproblem capable of generating a logical dual as this must be worked out on a problem-by-problem basis.

Implementing a Logic-Based Benders solver involves writing and solving models for a *master* and *slave* problem as well as writing glue code to move solutions and cuts between the two in an iterative pattern. Although most of the work that goes into this technique is the modeling of the master and slave problems themselves, there are significant advantages to a generic automated approach. In particular, Logic-Based Bender's is most often used in the context of an Integer Programming *master problem* and a CP *slave problem*. High level modeling of both the *master* and *slave* problems with transforms to reformulate to desired lower-level encoding can greatly reduce the modeling burden. Furthermore, the semantics of Logic-Based Benders are typically obscured in a handwritten solver where various solver APIs mix with custom glue code. Logic-Based Benders can be abstracted into a template pattern with generic glue-code provided under-the-hood allowing the user to focus only on what is really needed, that is, the high-level modeling.

Example: The *Warehouse Location-Allocation Problem* (WLAP) is a more complicated formulation of the *Warehouse Location* Problem (WLP) introduced in section 1.2.3. The original WLP simply chose locations to open warehouses and assigned stores to the warehouse

locations. The WLAP takes this a step further by assigning a fleet of trucks to each warehouse capable of servicing the stores each day. The WLAP is a difficult problem and complete CP and IP approaches have only been capable of solving modest instances. An IP model for the full problem was introduced in [6] and solved with a custom local search (TabuSearch). The problem was more recently reformulated for Logic-Based Benders (see [32]) providing a high performance complete solver.

Figure 3.5 shows the formulation of the problem adapted from [32]. The reader is encourage to refer to [32] as the problem description here will provide a brief summary of their models. The *master* is responsible for choosing warehouses to open, assigning stores to the warehouses and allocating a fleet of trucks for each warehouse. The *slave* is responsible for assigning individual trucks to specific stores. The *master* problem uses the following parameters and decision variables:

I : index of stores

J : index of warehouses

p_j : Boolean variables indicating whether warehouse j is open

x_{ij} : Boolean variables indicating whether store i is assigned to warehouse j.

f_j : fixed cost of opening warehouse j.

c_{ij} : cost of serving store i from warehouse j.

u : fixed cost of a truck.

t_{ij} : driving distance to serve store i from warehouse j.

$numVeh_j$: number of trucks allocated to warehouse j.

l : maximum daily driving distance of a truck.

b_j : service capacity of warehouse j.

d_i : service quantity consumed by store i.

$\bar{k}$: upper bound on the maximum number of trucks at any warehouse.

The objective of the IP *master* problem is to minimize the total cost of the warehouses, stores and trucks. Constraint (1) requires that each store be assigned to exactly one warehouse. Constraints (2) and (3) specify distance limitations. Constraint (4) limits the demand for each warehouse and constraint (5) defines the minimum number of trucks allocated to each facility.

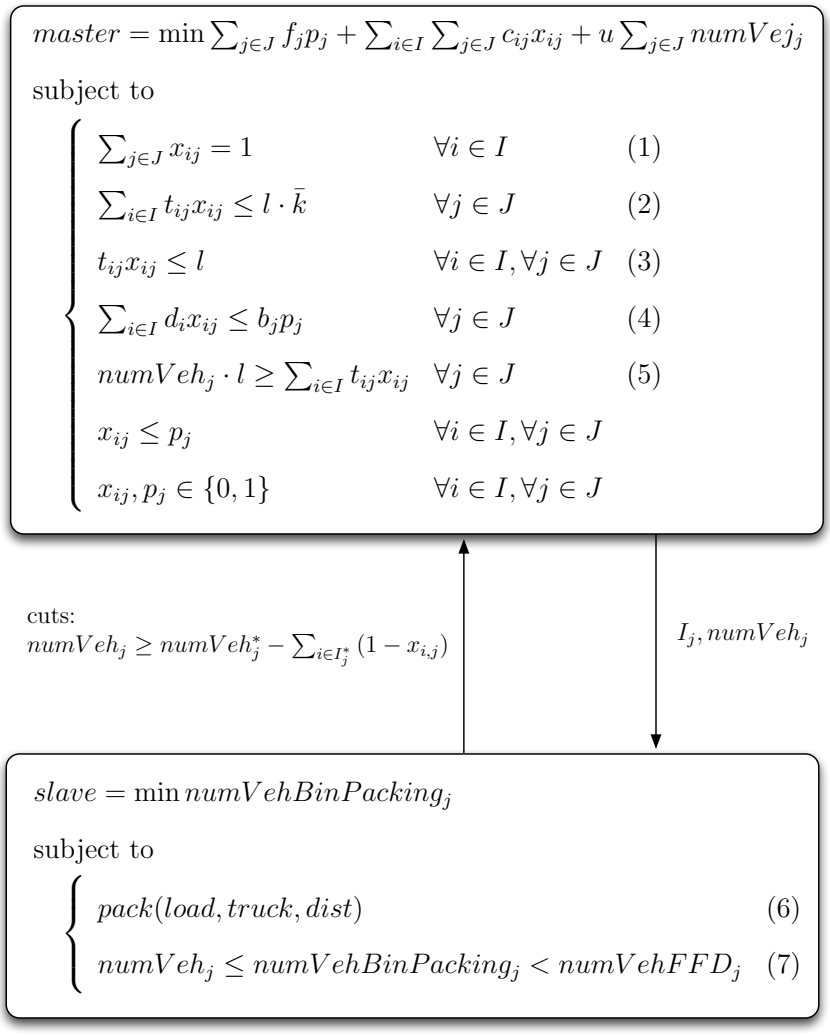

Figure 3.5: Logic-Based Benders formulation of the Warehouse Location Allocation Problem.

The CP *slave* problem is a collection of independent bin packing problems corresponding to each open warehouse. The *slave* receives from the *master* a set of stores assigned to each open warehouse (I_j) and a number ($numVeh_j$) of allocated trucks. The goal is

to assign stores to trucks while respecting the constraints on total travel distance. It makes use of the *global constraint* '*packing*' [83], which solves a bin packing problem. The variable array *load* has length equal to $numVehBinPacking_j$ which is the number of trucks required by warehouse j to service its stores. Each variable $load[k] \in \{0 \dots l\}$, for $k \in \{1 \dots numVehBinPacking_j\}$. The variable array *truck* is indexed over I_j and captures the truck assigned to each store. The constant vector *dist* describes the distance from each store in I_j to warehouse j. The constant $numVehFFD_j$ is an upper bound on the number of trucks produced by the first-fit decreasing heuristic (refer to the original paper [32]).

The procedure begins by solving an iteration of the *master* problem and then creating the *slave problems* for each of the open warehouses. The slaves are constructed with their own set of stores I_j and fleet of trucks ($numVeh_j$). The *slave* problem begins by finding the bound $numVehFFD_j$. If $numVehFFD_j = numVeh_j$, then *slave* is feasible and may terminate without solving the the bin packing problem. Otherwise, the *slave* tries to find the smallest value of $numVehBinPacking_j$ in the range $numVeh_j \dots numVehFFD_j$ that allows the bin packing to be satisfied. If the value of $numVehBinPacking_j$ turns out to be greater than $numVeh_j$, then a cut is produced: $numVeh_j \geq numVeh_j^* - \sum_{i \in I_j^*} (1 - x_{i,j})$. Here $numVeh_j^*$ and I_j^* are not variables, but the values from the solution in the previous iteration of the *master*. For a discussion of the intuition behind this cut, refer to [32]. Note that in this example, a *slave* is solved for each open warehouse location and a cut is generated for each infeasible *slave* problem. If all the *slave* problems are feasible, the Logic-Based Bender's method terminates, otherwise the process repeats by resolving the *master*. Instances of this problem will be solved using an automated Logic-Based Benders solver in Chapter 5.

3.5 Portfolio Solvers

Portfolio solvers have proven to be highly effective *hybrid* techniques over the past decade (see SATzilla [101]). Typically associated with SAT, CP and CP/SAT hybrids have become

more common in the past several years (see CPHydra [62], Sunny-CP [8] and Proteus [46]). The basic idea of a portfolio solver is to run a bunch of different solvers on the same problem and hope that one of the solvers is well-formulated to provide a quick solution. Hence, portfolio solvers encompass a broad array of techniques that can differ substantially in their implementation and the literature on these solvers has grown quite broad. The underlying similarity that defines a portfolio solver is the recognition that it is often difficult or impossible to know what solver will be effective on any given instance and, hence, a 'portfolio' of solvers is run on the problem in hopes that one of them works. Two major decisions define a portfolio solver:

1. What solvers should be in a 'portfolio.'

2. How should the solvers interact with one and other.

The first question has received a great deal of research attention with some modern techniques choosing among CP and SAT formulations using sophisticated machine-learning algorithms based on instance clustering and a k-nearest neighbor approach [62], [8], [46]. Tools for choosing a solver portfolio are not presented in this thesis, but certainly could be incorporated into the framework in future work.

The typical method of dealing with the second question has been to isolate the solvers and run them independent of one-and-other in a competitive fashion. This has been done both by providing all solvers equal *CPU* time as well as by weighting solvers and providing more promising solvers with more time on the *CPU*. This thesis argues for a different approach to solver interaction, that is, cooperation rather than competition. In particular, the tools presented in this thesis are very well positioned to allow for seamless development of a portfolio of cooperative solvers running in parallel across several technologies. The difficulties in developing this kind of cooperative solver are similar to those of parallel hybrids (refer to section 3.2). Cooperative portfolio solvers will be covered in detail in Chapter 6.

3.6 Lagrangian Relaxation

Lagrangian relaxation (e.g., [3]) is a commonly used optimization paradigm that typically applies to models that feature both easy and hard constraints. Lagrangian relaxation is coved in most textbooks on Integer Programming, including [61] and [24]. The idea is to relax the hard constraints into the objective leaving only easy constraints for the solver to deal with. A vector of lagrangian multipliers, or weights are used to adjust the 'penalty' associated with violating these relaxed constraints. Consider, for instance, the application of lagrangian relaxation to a mixed integer program with both easy constraints ($A_e x \geq b_e$) and hard constraints ($A_h x \geq b_h$):

$$Z = \texttt{min } c \cdot x \qquad\qquad Z_{LR}(\lambda) = \texttt{min } c \cdot x + \lambda^T \cdot (b_h - A_h x)$$
$$\texttt{s.t.} \begin{cases} A_h x \geq b_h \\ A_e x \geq b_e \\ x \in \{0,1\} \end{cases} \longrightarrow \texttt{s.t.} \begin{cases} A_e x \geq b_e \\ x \in \{0,1\} \\ \lambda \geq 0 \end{cases}$$

The relaxed problem, $Z_{LR}(\lambda)$, is called the Lagrangian Dual of Z with respect to A_h. Notice that only the easy constraints remain as the hard constraints have been relaxed. The vector $b_h - A_h x$ appearing in the objective provides a measure of the degree of violation or satisfaction of the relaxed constraints. The following properties hold for each relaxed constraint:

1. $b_h^i - A_h^i x \leq 0$ whenever the relaxed constraints is satisfied.
2. $b_h^i - A_h^i x > 0$ whenever the relaxed constraints is violated.
3. A constraint is said to be *tight* whenever $b_h^i - A_h^i x = 0$.
4. Whenever $b_h^i - A_h^i x < 0$, the constraint is said to have *slack* equal to the difference $|b_h^i - A_h^i x|$.

Hence, $Z_{LR}(\lambda) \leq Z$ for $\lambda \geq 0$, implying that $Z_{LR}(\lambda)$ is indeed a *relaxation* of Z.

```
1  function SubgradientMethod(Z_LR, Z̄)
2          π = 2
3          k = 0
4          λ_0 = 0⃗
5          Z_best = −∞
6          noImproveCount = 0
7          do
8                  x^{k+1} = solve(Z_LR(λ_k))
9                  Z^{k+1} = f(x^{k+1}) + Σ_{h∈H} λ^h_k · σ_h(x^{k+1})
10                 Δ^{k+1} = π(Z̄ − Z^{k+1})/||σ(x^{k+1})||^2
11                 forall (h ∈ H)  λ^h_{k+1} = max(0, λ^h_k + Δ^{k+1} * σ_h(x^{k+1}))
12                 if Z^{k+1} > Z_best
13                         Z_best = Z^{k+1}
14                         noImproveCount = 0
15                 else noImproveCount = noImproveCount + 1
16                 if noImproveCount > 30
17                         π = π/2
18                         noImproveCount = 0
19                 k = k + 1
20         while the termination criterion is not met;
21         return Z_best
```

Figure 3.6: Lagrangian Relaxation with the *subgradient* method.

Lagrangian relaxation is an iterative process which attempts to find optimal weights λ in $Z_{LR}(\lambda)$ to get as tight a bound as possible on Z. This is achieved by minimizing the *duality gap*, defined as $Z - Z_{LR}(\lambda)$. At each iteration the weights must be updated to permit a tighter bound, this is typically done with some variant of the *subgradient method* (see Figure 3.6 or textbook [24]). Note that the subgradient method requires an upper bound on the objective of Z (denoted $\bar{Z}$) which must be obtained by some other means. Generally, the convergence rate of the subgradient method is related to the quality of $\bar{Z}$. At a high level the procedure carries out the following basic steps:

1. Choose initial weights λ_0 (often set to 1 or 0).

2. At iteration k, solve $Z_{LR}(\lambda_k)$.

3. If the solution is also feasible with respect to Z, then finish.

4. Otherwise, generate weights λ_{k+1}, increment k and return to step 2.

As Lagrangian Relaxation progresses, the *duality gap* shrinks producing increasingly tight bounds on the original problem, Z (see Figure 3.7). The *subgradient method* is known to

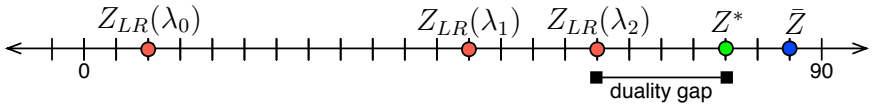

Figure 3.7: Convergence of Lagrangian Relaxation through three iterations, $Z_{LR}(\lambda_0)$, $Z_{LR}(\lambda_1)$ and $Z_{LR}(\lambda_2)$. The optimal solution Z^* is typically unknown and the upper bound $\bar{Z}$ is used by the subgradient method to update λ at each iteration.

converge to a *duality gap* of 0. In practice, however, Lagrangian Relaxation is often run on very difficult problems in which the *duality gap* cannot be fully closed within an acceptable running time. Early iterations of lagrangian relaxation typically are solved quickly as the solver pays little penalty for violating the relaxed constraints and, hence, can largely ignore them. As λ is refined and the 'penalty' grows for violating the hard constraints, solving the Lagrangian Dual at each iteration becomes much more difficult and the running time can quickly increase. Chapter 8 will discuss LR in greater details as well as providing a generalization of the method and some case studies.

Chapter 4

Literature Review

4.1 G12 Project

The G12 project (see [28], [70]) was an early attempt at building a high-level platform for solving *discrete optimization problems* through problem reformulation and automation of hybrids. The idea was to mix a high-level modeling language with an annotation language allowing a user to model a problem and then add annotation indicating how the model should be transformed or mapped to a lower level solver. G12 consisted of several components:

1. Zinc - A high-level, technology-independent modeling language including data-structures (arrays, sets, sequences) , loops, functions, predicates and reification

2. Cadmium - A mapping language specifying how an abstract *Zinc* model should be mapped to a technology specific model. In the case of a hybrid-solver, Cadmium also allows specification of communication between different solvers.

3. Mercury - A lower level language which runs the resulting problem.

G12 takes quite a different approach from what is proposed in this thesis. In particular, models are not first-class-objects in G12, and cannot be arbitrarily manipulated and combined by a chain of generic model transformations. Rather, the user models the problem

once in a high-level modeling language (*zinc*). A second 'annotation' language (*cadmium*) is then applied to the high-level model describing how the user would like the model mapped to a low-level solver (*mercury*). This process can be seen in an example taken from [28] for the *cutting stock problem*. For comparison refer to section 5.1.3 to see how this problem was solved in *CML* and finally the *column generation combinator* provided by Objective-CP in section 5.4.4. Figure 4.1 shows *cutting stock* modeled in *zinc*/*cadmium*. The semantic separation of the master and subproblem is not made clear and the model heavily relies on cadmium annotations to indicate which variables belong to the *master problem* and *subproblem*. The model also includes explicit annotations for linking constraints requiring the user to describe how columns are passed back into the master problem.

Although the approach taken by G12 has advantages, such as flexibility in mapping from high-level to low-level representations, it is not an approach which is conducive to building up arbitrarily complex hybrids. Furthermore, it does little to improve the accessibility of advanced hybrid techniques to non-expert users. The effective use of *cadmium annotations* for building hybrids requires a detailed technical understanding of the low level algorithmic details employed by the hybrid technique.

G12 has also been successfully applied to branch-and-price hybrid models, see [70].

CuttingStock.zinc

```
int: K;                                type Pieces = 1..K :: colgen_symmetric;
int: N;                                type Items  = 1..N;
int: L;
array[Items] of int: i_length;         array[Items] of int: i_demand;

array[Pieces]        of var 0..1: pieces :: colgen_var;
array[Pieces, Items] of var int:  items  :: colgen_var;

solve :: lp_bb([pieces, items], most_frac, std_split)
      :: colgen_ph(100, 10) :: colgen_solver("lp")
    minimize sum([ pieces[k] | k in 1..K]);

constraint forall(i in 1..N)(sum([items[k, i] | k in 1..K]) >= i_demand[i]);

constraint forall( k in 1..K)(
               sum(i in 1..N)(items[k,i] * i_length[i]) <= pieces[k] * L
                   :: colgen_subproblem_constraint(k, "knapsack"));
```

Figure 4.1: A Zinc model for *cutting stock* using cadmium annotations.

4.2 SIMPL and Search-Infer-Relax

Search-Infer-Relax is a framework proposed by Hooker in [44] and implemented as SIMPL [102]. *Search-Infer-Relax* is a conceptual framework demonstrating that many optimization techniques in the literature can be abstracted into special cases of a single solution method (called *Search-Infer-Relax*). The idea is that by abstracting optimization techniques in this way, they can be brought 'under one roof' and mixed and matched seamlessly. Many optimization techniques are composed of three fundamental phases (these descriptions are taken from [44]):

Search Enumerates problem restrictions. Examples include, branching, generating subproblems, or local neighborhood search.

Inference May filter the search space, produce nogoods or cutting planes.

Relaxation Places bounds on the objective function, reducing the search space.

The framework demonstrates that common CP techniques, local search heuristics and Logic-Based Bender's Decomposition can captured within this abstraction (among numerous

other techniques) and may be integrated into new hybrid techniques in interesting ways. These include continuous relaxation for global constraints, integration of CP and IP (using Logic-Based Benders) and relaxation bounds in local search.

4.3 Essence and Conjure

Essence and Conjure [41], [4] set out to address the difficulty of effective problem modeling. As was discussed in the introduction, the effectiveness of a solver is crucially dependent on how well a problem is modeled for a particular technology. Essence is a language for specifying a problem combinatorially at a level of abstraction above which most modeling decisions are made. *Essence* is then combined with *Conjure* [5] to automate the generation of a constraint model using rewrite rules derived from 'expert knowledge' maintained in a database. This work has demonstrated that it is possible in many cases to automate the modeling step for CP and achieve results comparable to hand-written 'expert' models. Hence, Essence + Conjure provide an automation platform orthogonal to what is proposed in this thesis.

4.4 Z3 SMT Engine

Recent work from Microsoft Research on SMT solvers [26] has similar objectives to this thesis. The *Z3* SMT engine allows user to specify *tactics* which are user specified expressions used to direct the search procedure. *Tactics* are capable of relaxing parts of the SMT problem and then determining whether a particular relaxed subproblem will provide an upper or lower bound for the original problem. These tactics can be queried at runtime to determine how the search should proceed and which tactics should be invoked next. In this sense, the tactics in *Z3* function in a similar manner to model operators and metadata presented in this work.

Chapter 5

A Framework for Automated Hybrid Generation

This chapter will lay out a new framework for generic model manipulation, automation of complex hybrids and semantic validation of model composition. The motivation for this work is presented in Chapter 2. The first section presents an early prototype system called *CML*, a custom modeling and composition language built on top of the COMET platform.

5.1 Prototype: Comet & CML

COMET [89], [55] is a full-featured programming language designed as a unified environment for solving combinatorial optimization problem using various techniques including *Constraint Programming*, *Mixed-Integer Programming*, and *Constraint-Based Local Search*. COMET provides syntax for declarative modeling and for specifying a search procedure as well as consistent APIs for working with various solver technologies. By making various solver technologies available from within the same programming language with similar APIs, COMET simplifies the process of writing *hybrid solvers* as well as the difficulty of moving between solver technologies. COMET has a syntax similar to Java and C++ and builds on the work done in earlier platforms such as *OPL* [90]. COMET also provides features such as *closures*

and *continuations* as first-class objects to facilitate the creation of non-deterministic search procedures as well as abstractions for parallel-solving based on search-space splitting and work-stealing.

From the point of view of this thesis, several of the major advantages of COMET are:

1. A common environment for building CP, MIP, LP and LS models, greatly easing the development of hybrids.

2. The state-of-the-art Local Search (CBLS, see [92]) implementation.

3. Custom search for CP and LS with out-of-the-box high-level search construct build directly into the language.

Early attempts at implementing high-level modeling along with operators for transforming models, concretizing and generating hybrids were done in a custom scripting language called *CML* (COMET Modeling Language). *CML* features models as generic first class objects that can be created, transformed, concretized into technology specific representations, and combined into hybrid solvers. *CML*, however, lacks any solving capabilities of its own and instead generates COMET files representing the final problem to be solved (often a hybrid solver). Some examples of *CML* and COMET's abilities are shown in the following sections.

5.1.1 Parallel Hybrid: MIP & LS

A bounds sharing parallel hybrid solver (refer to section 3.2) is typically difficult to implement in practice, particularly when interfacing disparate solver technologies. In the following example, *CML* allows a MIP solver to be combined in parallel with a *constraint-based local search* (CBLS) solver. These technologies had never been combined into a parallel solver previously, but *CML* makes it possible with just several lines of code.

Consider the *assignment problem* (first introduced in section 1.2.1) in *CML* (Figure 5.1). The problem is modeled in a compact, technology agnostic format in lines 1-10. Lines 2-6

define problem constants, including a cost matrix C. Line 7 introduces an array of decision variables representing *agent* assignments. Line 8 specifies the objective, minimizing the total cost of the assignments. Line 9 posts an *alldifferent* constraints requiring agents to be assigned to distinct *tasks*. Line 11 applies several operators to the generic *Assignment* model creating a new hybrid model. In particular, the *Assignment* model is passed into two *concretization* operators *LS* and *MIP*. These operators transform the generic model into two technology specific representations, called *concrete models*. The two concrete models are then combined in a bounds sharing parallel hybrid solver with the *parallel_compose* operator. The end result is emitted as a COMET file which captures the desired hybrid (Figure 5.2) (for an overview of COMET syntax and capabilities, see [55]). Note that, despite COMET's high-level capabilities, the generated hybrid still requires that low-level details such threads and notification callbacks be setup to permit communication and solver termination. The COMET code illustrates the work that would be required to write a parallel hybrid by hand. By leveraging *CML*, the user is able to avoid such low level details and instead automate the hybridization process.

```
1 model Assignment {
2       n = 50;
3       agents = n;
4       tasks = n;
5       dist = UniformDistribution(1..20);
6       C = [1..agents, 1..tasks: dist.get()];
7       var{int} A[1..tasks](1..agents);
8       objective: Minimize(sum(t in 1..tasks) C[A[t], t]);
9       post: AllDifferent(A);
10 }
11 hm = parallel_compose([LS(Assignment),MIP(Assignment)]);
12 hm.emit_comet_file("Assignment.par_hybrid.co");
```

Figure 5.1: Assignment Problem in CML.

Notice that the model is specified in a high level abstract way and then concretized using two different technologies, MIP and LS. The MIP concretization requires reformulation as neither the objective function nor the *AllDifferent* constraint are linear. The reformulation is done using a *ModelLinearizer* operator which is coded entirely in *CML* and is part of *CML*'s standard library. The advantage of having *CML* operators written directly in *CML* is

```
1 import CMLNotificationCenter;
2 CMLNotificationCenter nc();
3 import cotls;
4 include "genericLocalSearch";
5 import cotln;
6 Model<LS> lsm();
7 Solver<MIP> ipm("SCIP");
8 whenever nc@newBest(int best) ipm.getObjective().setPrimalBound(best);
9 int C[1..30, 1..30] =   ... // initialization of C
10 {
11    var{int} A[1..30](lsm.getLocalSolver(), 1..30);
12    expr{int} __e0[1..30] = [A[1], A[2], A[3], ..., A[30]];
13    lsm.minimizeObj(C[A[1], 1] + C[A[2], 2] + ... + C[A[30], 30]);
14    lsm.post(alldifferent(__e0));
15    lsm.close();
16 }
17 TabuSearch tabuSearch(lsm);
18 whenever tabuSearch@localBest(Solution s,int fBest) notify nc.newBest(fBest);
19 in {
20    var<MIP>{int} __c30[1..30, 1..30](ipm, 0..1);
21    var<MIP>{int} A[1..30](ipm, 1..30);
22    var<MIP>{int} __A0[1..30, 1..30](ipm, 0..1);
23    minimize<ipm> 15*__A0[1, 1] + 1*__A0[2, 1] + ... + 9 * __A0[30, 30]
24    subject to {
25            ipm.post(__A0[1, 1] + __A0[2, 1] + ... + __A0[30, 1] == 1);
26            ... // Many more constraints
27    }
28    using {
29       thread t1 { tabuSearch.apply(); }
30       ipm.defaultMinimize();
31       notify nc.terminate();
32    }
33 }
```

Figure 5.2: Generated COMET Model for the Assignment Hybrid.

that users can easily extend or completely replace the built in operators to perform custom reformulation of models. Listing 5.1 below shows the method used to reformulate *AllDifferent* constraints.

This method listing gives a sense of the syntax of *CML* which is a mix of COMET and dynamic scripting languages such as *Python*. The method *apply_alldifferent* is part of an operator class that is building a new linear model from a given abstract model. The linear model is stored in the instance variable *LM*. Line 2 declares a new two dimensional variable array with an automatically generated unique name. This model variable will represent the linearization of the *AllDifferent* constraint and is stored in the CML variable x. A sharp reformulation of *AllDifferent* is well known and is described in [73]. The *forall* in lines 4-7 restricts the linearization of each variable expression such that it must take on exactly one value from the domain. The second *forall* (lines 9-14) steps through the variable expressions

Listing 5.1: Linearization of *AllDifferent*

```
1 def apply_alldifferent(alldiff) {
2         x = LM.declare_var_array(unique_name, [alldiff.range, alldiff.domain], 0..1);
3
4         forall(i in alldiff.range) {
5                 expr = (sum(j in alldiff.domain) x[i, j]) == 1;
6         LM.post_constraint(expr);
7         }
8
9         forall(i in alldiff.range) {
10                item = alldiff[i];
11                if(!item.is_linear) item = ExprLinearizer.apply(item);
12                expr = (sum(j in alldiff.domain) j * x[i, j]) == item;
13        LM.post_constraint(expr);
14        }
15
16        forall(j in alldiff.domain) {
17                expr = (sum(i in alldiff.range) x[i, j]) <= 1;
18        LS.post_constraint(expr);
19        }
20 }
```

in the *AllDifferent* array linearizing any non-linear expressions and then constraining the linearization to be equal to the value of the variable expression. The final *forall* (lines 16-19) enforces the all different requirement by constraining the linearization to have at most one variable expression assigned to each domain value.

The objective function is linearized using the *ExprLinearizer*. The *ExprLinearizer* class performs a traversal of an expression tree allowing non-linear parts of the expression to be replaced by appropriate linear reformulations. In this case, the expression $C[A[t], t]$ is non-linear. The linearization of this type of expression is also well known, but more involved than the reformulation of *AllDifferent*. The implementation is fully generic and model independent and can be found in the CML standard library. The generated COMET code is shown in figure 5.2, and has been abbreviated as the actual output is too large to include.

The concretizations, by default, make use of common blackbox search heuristics. In the case of MIP, the search is provided by the underlying MIP engine (e.g. SCIP), while LS uses a generic TabuSearch. It is of course, possible to use an alternative blackbox search or provide a block of COMET code specifying a custom search. The two concretizations are composed in parallel and a hybrid COMET model is generated. The hybrid search is substantially faster than a MIP or CP search alone as the LS is able to provide a good upper

bound.

It is important to note that LS-MIP bound passing hybrids have not been previously possible in high level languages such as COMET or *Zinc* as there has been no mechanism for passing bounds from the LS into the MIP backend (in this case SCIP) during the branch-and-bound search. The work required to setup SCIP callbacks to pass bound change events mid-search is quite substantial and, hence, LS-MIP bound passing hybrids are generally not done. For this work, SCIP callbacks were created to allow COMET's event system to work seamlessly with SCIP. *CML* makes it easy for the end user to compose MIP and LS searches in parallel, automatically generating the appropriate COMET events while COMET communicates in real time with the SCIP branch-and-bound search.

5.1.2 Sequential Hybrid: LS & CP

Sequential hybrids (section 3.1) are simpler than parallel hybrids and are used quite often in practice. Despite being easier to implement than a parallel hybrid, automation with *CML* still provides substantial benefits. The Warehouse Location Problem (WLP) was first introduced in section 1.2.3. Despite being relatively simple to model, the problem quickly becomes intractable for complete CP or MIP searches. On the other hand, well formulated CBLS models can produce high quality solutions quickly, but lack the ability to prove optimality. *CML* allows a sequential hybrid CBLS-CP approach with very good results and minimal efforts.

Figure 5.3 shows the CML model. Lines 1-11 read in instance specific data for the problem. Lines 13-17 model the problem, introducing variables representing open warehouses in line 14 and posting an objective in lines 15-16. Line 19 generates the sequential hybrid with a chain of several operators. The high-level *warehouse* model is concretized into technology specific instances by the *CP* and *IP* operators. The LS model is then transformed into a time limited solver that will run for 5 seconds by the *TimeLimit* operator. Finally the CP and time limited LS models are composed in sequence by the *sequential_compose* operator.

```
1 f = File("data.txt", "r");
2 n = f.read_integer();
3 m = f.read_integer();
4 warehouses = 1..n;
5 stores = 1..m;
6 fcost = [warehouses: 0]; # Create array of zeros
7 tcost = [warehouses, stores: 0]; # Create matrix of zeros
8 forall(w in warehouses) {
9     fcost[w] = f.read_integer(); # warehouse opening cost
10    forall(s in stores) tcost[w, s] = f.read_integer(); # transportation cost
11 }
12
13 model Warehouse {
14    var{int} open[warehouses](0..1);
15    objective: Minimize((sum(w in warehouses)fcost[w] * open[w])
16                        + sumMinCost(open, tcost));
17 }
18
19 shm = sequential_compose([TimeLimit(LS(Warehouse), 5000), CP(Warehouse)]);
20 shm.emit_comet_file("Warehouse.seq.co");
```

Figure 5.3: Warehouse Location Problem in CML.

Like COMET [89], *CML* uses the global function *SumMinCost* to model the variable cost part of the objective. The *SumMinCost* expression is defined as follows

$$SumMinCost(v, C) = \sum_{j \in stores} C[argmin_{\{i \in warehouses | v[i]==1\}}(C_[i, j]), j]$$

and is parameterized by a Boolean variable array v denoting *open* facilities and a cost matrix C. The matrix C has a row for each warehouse and its columns correspond to stores. An entry $C[i, j]$ indicates the cost of supplying store j from warehouse i. The *SumMinCost* function, then, maintains the minimum cost of supplying all the stores given the array of open warehouse locations. *SumMinCost* provides an example of a *CML* concretization rule for a high level modeling structure. The concretization into CBLS is nearly a one-to-one mapping. The finite-domain solver of COMET does not currently support *SumMinCost*. Hence, during the concretization process, *CML* reformulates *SumMinCost* with a suitable template generating new variables and constraints with equivalent semantics. The source-to-source concretization template (not actual *CML* code) *CML* uses for *SumMinCost* is:

```
1 SumMinCost(var v[R0](D),int c[R0,R1]) ⇔ α
2 with
3         var{int} sc[R1](0..max(i in R0,j in R1) c[i,j]);
4         var{int} ws[R1](R0);
5         var{int} α(0..max(i in R0,j in R1) c[i,j]);
6 in {
7     forall(i in R1) {
8        post : sc[i] == c[ws[i],i];
9        post : v[ws[i]] == 1;
10        }
11        post : α == sum(i in R1) sc[i];
12 }
```

Namely, when *CML* concretizes the abstract model for a CP solver, it introduces auxiliary variables and additional constraints in the concrete model to ensure that a fresh variable α can be substituted for *SumMinCost*. Specifically, it declares an array of variables *sc* (line 3) meant to hold the cost of each store in R_1. It declares an array of variables *ws* (line 4) meant to track the warehouse assigned to each store in R_1 (Hence its domain is R_0, the range of v). It finally declares a variable α that will hold the value *SumMinCost*. The template introduces constraints in line 8-9 to define the store cost and link the warehouse selection to v. Line 11 of the template constraints α to equate with the total store cost. *SumMinCost* can then simply be replaced by α in the objective.

The result of the sequential composition is a hybrid COMET model which runs a CBLS solver for 5 seconds, then passes the best bound on the objective to a complete CP search. These models were tested on a randomly generated large instance. A pure CP model takes nearly 8 hours to terminate while the hybrid CBLS-CP solver finishes in just over 20 minutes. From Figure 5.3 it is clear that the 'hardest' task is to read the data. The Warehouse problem is stated concisely in a few lines and the hybrid solver is generated with a single line.

5.1.3 Column Generation

CML also extends naturally to template-based hybrids such as *column generation* (refer to section 3.3). Consider the example of *Cutting Stock* in *CML* (Figure 5.5). Column Generation is made into a generic hybrid in *CML* through the use of *templates* and a dictionary mapping to define how the dual values fit into the *slave problem* and how to inject solutions back into the *master*.

CML implements the column communication channel through COMET events. The approach makes it possible to easily adopt custom code for column definition and injection. CML supports the use of quoted blocks of COMET code (inserted between forward tick marks ‘) and these blocks may be stored in *CML* variables. The following fragment illustrates how to emit a COMET block to achieve a custom column injection:

```
1 whenever cgm@injectColumn(col) `
2         masterInject(col); // Arbitrary manipulation of subproblem solution.
3 `
```

Such functionality is useful when solutions to the subproblem do not easily map to master problem columns. Note that column generation models can already be directly implemented with COMET and an example of the cutting stock problem can be found in the COMET distribution. Writing such a model, however, remains time consuming and error prone as the syntax for injecting a column and mapping duals remains delicate. Most of the *glue* code required for column generation can be abstracted out and turned into a template. Hence, column generation in CML is implemented by first concretizing the *master* and *slave* problems and then plugging these concretizations into the underlying COMET template. Figure 6 shows the template.

Template: The template takes as arguments a triple $\langle master, slave, mapping \rangle$ and an output stream f. The :: operator is being used to indicate block and statement concatenation into the stream and the code being emitted to the output block is highlighted in blue to

```
1 def ColumnGeneration(<master, slave, mapping>, f) {
2         if(master.type != LP) error
3         col_vars = mapping.master.variables;
4         col_coef = mapping.master.columns;
5         if(col_vars.size != col_coef.size) error;
6         dual_vars = mapping.slave.duals;
7         if(!dual_vars.size != master.constraints.size) error
8         column_vars = colgen.mapping.slave.column;
9         if(column_vars.size != master.constraints.size) error
10        # Emit Master
11        f = f::master.declarations;
12        f = f::master.variables;
13        f = f::master.model;
14        # Emit Slave
15        f = f::do {;
16        # Perform mapping of duals
17        forall(i in dual_vars.size)
18                f = f::dual_vars[i] = master.constraint[i].dual;
19        f = f::slave.declarations;
20        f = f::slave.variables;
21        f = f::slave.model;
22        # End when new column cant improve objective
23        f = f::if(slave.objective >= 0) break;
24
25        # Inject column into master and repeat.
26        f = f::else notify master.injectColumn(column_vars);
27
28        f = f::while(true);
29 }
```

Figure 5.4: Column Generation Template

distinguish it from the logic of the template. First notice that the template is performing model verification using metadata about the concretization technology from the *master* and *slave* models. Line 2 verifies that the *master* has been concretized using an LP relaxation and line 5 verifies that the number of column variables matches the number of coefficients. Lines 7 and 9 insure that the variables (in the *slave*) being mapped to the duals and the variables in the *slave* which will be injected into the *master* (as a new column) match the number of constraints in the *master*. The template also shows how the mappings are used to *glue* the models together. For example, lines 6 uses the *mapping* to grab the *slave* variables that should be mapped to the duals and then actually fixes them on lines 18-19 using the dual values obtained from the *master* constraints.

Despite the fact that the implementation of Cutting Stock in COMET is not particularly large, the complexity of the syntax really hides the underlying semantics of column generation and makes the model prone to bugs. Similar criticism could be made of column generation in G12 (refer to the literature review in section 4.1) as the model is laden with complex

```
board_width = 110;
shelves = 1..5;
shelf = [shelves : 20, 45, 50, 55, 75];
demand = [shelves : 48, 35, 24, 10, 8];
columns = [];
# Create initial set of columns
forall(i in shelves) {
    col = [shelves : 0];
    col[i] = floor(board_width / shelf[i]);
    columns.append(col);
}

model CuttingStock {
      var{int} cut[columns.range()](0..demand.max());
      objective: Minimize(sum(i in cut.range()) cut[i]);
      forall(i in shelves)
         post: sum(j in cut.range() columns[j, i] * cut[j]) >= demand[i];
}

model Knapsack {
    var{int} use[shelves](0..board_width);
    var{int} cost[shelves](-100..100);
    objective: Minimize(1 - sum(i in shelves) cost[i] * use[i]);
    post: sum(i in shelves) shelf[i] * use[i] <= board_width;
}

mapping = Dictionary();
mapping["master.columns"] = columns;
mapping["master.variables"] = CuttingStock.cut;
mapping["slave.duals"] = Knapsack.cost;
mapping["slave.column"] = Knapsack.use;
cgm = ColumnGeneration(LP(CuttingStock), CP(Knapsack), mapping);
cgm.emit_comet_file("CuttingStock.co");
```

Figure 5.5: Cutting Stock Problem in CML

annotations required to direct the generation of concrete solver. Column generation in CML abstracts away the more difficult aspects of actually implementing column generation and really brings the semantics of the model to the forefront. On the other hand, the *metadata* available to checking semantics is still limited in *CML* as will be discussed later.

5.1.4 LNS

CML operators allow for easy generation of solvers for Large-Scale Neighborhood Search (LNS, see section 1.5.1). LNS algorithms have proven to be very effective for certain applications, yet remain time consuming to write. Finding effective subset of variables often devolves into a trial-and-error approach.

CML provides three LNS operator which allow users to implement several blackbox LNS heuristics or specify completely custom code. At its core, a large neighborhood process

```
1 import cotfd;
2 import cotln;
3
4 class Column {
5 var<LP>{float} __var;
6 int[] __coef;
7 Column(var<LP>{float} v, int[] coef) { __var = v; __coef = coef; }
8 var<LP>{float} variable() { return __var; }
9 int getCoefficient(int i) { return __coef[i]; }
10 int[] getCoefficients() { return __coef; }
11 }
12 stack{Column} columns();
13 Constraint<LP> __c[1..5];
14
15 // Master
16 Solver<LP> lpm("clp");
17 var<LP>{float} cut[1..5](lpm);
18 int __coef0[1..5] = [5, 0, 0, 0, 0];
19 columns.push(Column(cut[1], __coef0));
20 int __coef1[1..5] = [0, 2, 0, 0, 0];
21 columns.push(Column(cut[2], __coef1));
22 int __coef2[1..5] = [0, 0, 2, 0, 0];
23 columns.push(Column(cut[3], __coef2));
24 int __coef3[1..5] = [0, 0, 0, 2, 0];
25 columns.push(Column(cut[4], __coef3));
26 int __coef4[1..5] = [0, 0, 0, 0, 1];
27 columns.push(Column(cut[5], __coef4));
28 minimize<lpm> cut[1] + cut[2] + cut[3] + cut[4] + cut[5]
29 subject to {
30 __c[1] = lpm.post(5 * cut[1] + 0 * cut[2] + ... + 0 * cut[5] >= 48);
31 __c[2] = lpm.post(0 * cut[1] + 2 * cut[2] + ... + 0 * cut[5] >= 35);
32 __c[3] = lpm.post(0 * cut[1] + 0 * cut[2] + ... + 0 * cut[5] >= 24);
33 __c[4] = lpm.post(0 * cut[1] + 0 * cut[2] + ... + 0 * cut[5] >= 10);
34 __c[5] = lpm.post(0 * cut[1] + 0 * cut[2] + ... + 1 * cut[5] >= 8);
35 }
36 // Subproblem
37 do {
38 Solver<CP> cpm();
39 float cost[i in 1..5] = __c[i].getDual();
40 var<CP>{int} use[1..5](cpm, 0..110);
41
42 minimize<cpm> 1 - (cost[1] * use[1] + ... + cost[5] * use[5])
43 subject to {
44 cpm.post(20 * use[1] + 45 * use[2] + ... + 75 * use[5] <= 110);
45 }
46 if (cpm.getObjective().getValue().getFloat() >= -0.00001) break;
47 else {
48 Column<LP> col(lpm);
49 col.setObjectiveCoefficient(1.0);
50 forall(i in 1..5)
51 col.setGeqConstraintCoefficient(__c[i].getId(), use[i].getValue());
52 var<LP>{float} n(lpm, col);
53 columns.push(Column(n, all(i in 1..5) use[i].getValue()));
54 }} while(true);
```

Figure 5.6: Generated COMET Model for the Cutting Stock.

requires the specification of three processes, namely: (1) How to partition the variables that are going to be searched over versus the variables that are frozen, i.e., choosing the active and the frozen fragments; (2) Freeze the variables in the frozen fragment to specific values;

and (3) Search over the variables in the active fragment.

Random LNS. The Random LNS (`RndLNS`) operator lets a user specify a fragment size (and optionally a search block) and then generates an active fragment (and its 'to be frozen' complement) by selecting active variables at random and freezing all remaining variables with random values from their domains. If a search block is not specified, `RndLNS` will pick up the current search heuristic and use it for searching over the fragment. The example below specifies the default search as IBS

```
1 cpm = RndLNS(IBS(CP(Radiation)));
```

Propagation Guided LNS. The PGLNS (`PGLNS`) operator offers a Propagation Guided Large-Scale Neighborhood Search (PGLNS) [66]. PGLNS is a high performance black-box LNS search that automatically generates fragments using propagation metadata. Given a concrete CP model, `PGLNS` can be used without further ado, yet it can also be refined with a customized search block. See the example below:

```
1 cpm = PGLNS(IBS(CP(Radiation)));
```

Standard LNS. Standard LNS (`StdLNS`) allows the greatest customization over large neighborhood search. The operator requires users to specify the freezing process for the to-freeze variables, and, optionally, the searching process over the active fragment. This is the operator used in the radiation therapy example in Chapter 7. Once again, CML relies upon the COMET event machinery to implement the process separation. Specifically, CML uses COMET events to trigger the execution of code responsible for each processes. For instance, line 27 in Figure 5.7 states that upon reception of the `freeze` event on the concrete model *cpm*, the specified quoted COMET block should execute to carry out the freezing. The example below illustrates the selection of standard LNS as well as the custom freezing process.

```
cpm = StdLNS(IBS(CP(Radiation)));
whenever cpm@freeze() ` forall(i in ${floor(bt\_max/2)..bt\_max}) label(N[i], 0); `
```

COMET blocks follow a couple of simple rules. First, anything contained in ${..} will be evaluated by the CML interpreter and then embedded in the COMET block. This is needed to reference CML constants as the emitted COMET file will only contain constant literals and no references to the constant names as they appeared in CML. Second, decision variables appearing in abstract CML models will be emitted to COMET with identical names, so referencing decision variables from a COMET block requires no additional work. Even when several models use identically named variables, the output is still correct as each model is emitted in a separate lexical scope in COMET.

```
import "lib/LNS";
// LOAD intensity matrix ...
bt\_max = (all(i in rows, j in cols) intensity[i,j]).max();
ints_sum = sum(i in rows, j in cols) intensity[i,j];
btimes = 1..bt_max;
# Pre-compute optimal beam-on time
beam_time = 0;
forall(i in cols) {
        v = intensity[i, 1] + sum(j in 2..n) max(intensity[i, j] - intensity[i, j-1], 0);
        if(v > beam_time) beam_time = v;
}
model Radiation {
   var{int} K(0..m*n);
   var{int} N[btimes](0..m*n);
   var{int} Q[rows, cols, btimes](0..m*n);
   objective: Minimize(K);
   post: beam_time == sum(b in btimes) b * N[b];
   post: K == sum(b in btimes) N[b];

   forall(i in rows, j in cols)
     post: intensity[i,j] == sum(b in btimes) b * Q[i,j,b];
   forall(i in rows, b in btimes)
     post:  N[b] >= Q[i,1,b] + sum(j in 2..n) max(Q[i,j,b] - Q[i,j-1,b], 0);
}

cpm = StdLNS(IBS(CP(Radiation)));
whenever cpm@freeze() ` forall(i in ${floor(bt_max/2)..bt_max}) label(N[i], 0); `
cpm.emit_comet_file("Radiation.CP.co");
```

Figure 5.7: IMRT counter model in CML with LNS search

Custom Search and Active Fragment. If a user wishes to specify a custom search, a quoted COMET block must be provided. CML relies on COMET events to realize the

communication. All that is required is the specification of a COMET block to be executed upon reception of the *searchFragment* event as shown below:

```
1 whenever cpm@searchFragment() `
2    forall(b in ${btimes}: !N[b].bound()) {
3       while(!N[b].bound()) {
4          int mid = (N[b].getMin() + N[b].getMax())/2;
5          try<cpm> cpm.lthen(N[b],mid+1); | cpm.gthen(N[b],mid);
6 } } `
```

The quoted COMET block will be executed upon reception of the *searchFragment* event. Inside the quoted COMET block there is an expression ${btimes}. This expressions is evaluated by the CML interpreter and then embedded in the COMET block that is inlined into the concrete model.

The ability to quickly concretize a CP model and then *drop-in* various LNS search heuristics substantially reduce the overhead of developing a custom LNS. With the high level modeling tools provided in CML, LNS searches can be quickly built, tested and compared.

5.1.5 Shortcomings of the CML approach

CML was the first system to be built around the idea of capturing abstract models as first-class-objects for manipulation and arbitrary combination through *model operators*. The results of *CML* were very promising, but as the system developed, two major limitations emerged. The first limitation was the inability to sufficiently capture *semantics*. As this thesis has emphasized, solving problems using sophisticated multi-technology hybrids can be subtle. Although CML was capable of providing basic error checking, it lacked a mechanism for rigorously capturing the capabilities of a solver and inferring the meaning of the solutions and bounds being produced. This lead to a proliferation of operators which were very similar, but designed to operate on slightly different kinds of models. It was the responsibility of the user to know which flavor of operator was appropriate for their particular model. Furthermore, it was often possible to combine solvers for unrelated models in a manner

that produced a nonsensical hybrid without the system being capable of detecting a serious semantic error had occurred.

The second major limitation emerged from the fact that CML and COMET were completely separate languages. As efforts were made to incorporate semantics into *CML* as well as more sophisticated inter-solver communication. *CML* was expanded not simply to produce a COMET file, but to spawn independent COMET processes and facilitate their communication during the solving process itself. Keeping this scheme flexible relied on the use of COMET blocks within CML, which themselves became problematic. The COMET blocks allowed bits of COMET code to be passed through *CML* into the final COMET solver. The problem, however, was that *CML* spoke the language of a high level abstract model, while the final COMET model only understood a low-level technology specific encoding. Providing a clean mechanism for writing COMET blocks in *CML* became very cumbersome. Fortunately, the successor to COMET (OBJECTIVE-CP) was under development during this time period and many of the ideas of *CML* influenced the development of the new platform. The next several sections will discuss OBJECTIVE-CP and the libraries that were built on top of it to move beyond the shortcomings of *CML*.

5.2 High-Level Modeling in Objective-CP

5.2.1 Modeling & Concretizing

The OBJECTIVE-CP platform [94] is the successor to COMET, and as such, has many of the same features and advantages. Some of these similarities include:

1. Multi-technology environment, ideal for hybrid development.
2. State-of-the-art solver implementations, including: CP, LS and MIP (Gurobi wrapper).
3. Custom search implemented in host language.

The major difference is that many of the ideas developed in *CML* are built into the core of Objective-CP and expanded upon in many ways. In particular, models are technology independent, first-class objects in Objective-CP. Unlike Comet, users no longer write a CP model or a MIP model, but instead write a generic high-level model which can then be *concretized* into a technology specific *program* for solving. At a high level, Objective-CP has the following components:

Modeling : An abstract problem spec written at a high level and in technology agnostic way (similar to *CML*).

Transforms : Operators that map abstract models to other abstract models (linearization or relaxation for example).

Concretization : Operators that produce technology specific *programs* from an abstract *model*.

Search : Either a blackbox or user defined procedure for exploring the search space in terms of the abstract model.

These components will be formalized in the following sections.

5.2.2 Modeling Definitions

Definition 9 *A model M is of the form $\langle X, C, O\rangle$ where X is the set of model variables, C the model constraints and O the (optional) objective function.*

Definition 10 *A model transformation τ transforms a model $M = \langle X, C, O\rangle$ into another model $\tau(M) = \langle X_o, C_o, O_o\rangle$ satisfying $X \subseteq X_o$.*

Examples of model transformations performed by Objective-CP are shown in Figure 5.8. When models are in flattened form (sufficiently decomposed), they can be concretized in an optimization program.

Definition 11 *A model concretization γ takes a model M in a flattened form and concretizes M into a program $P = \langle M, \gamma \rangle$, where $P = \gamma(M)$. The concretization associates a concrete variable with every model variable, a concrete constraint to every model constraint, and a concrete objective with the model objective.*

To obtain an optimization program P from a model M, Objective-CP performs a series of model transformations followed by a concretization, e.g.,

$$P = \gamma(\tau_{k-1}(\cdots \tau_0(M) \cdots)).$$

Flatten	*Flattening* a model decomposes complex expressions into simpler ones, often adding variables and constraints in the process.
Continuous	Performs a continuous *relaxation* of a model replacing integer-valued domain constraints with continuous interval domains.
Linear	Creates a linear reformulation to replace global constraints and logical constraints with a set of equivalent linear constraints [72].

Figure 5.8: Examples of commonly used model operators.

The end result for users, is that low level details of technology specific encodings are abstracted away and the user only sees the high level model. While this may look like a limitation in some respects, it dramatically simplifies the process of reformulating models, specifying solver communication and writing a search as everything is captured in a single high-level representation. As an example, consider the *warehouse location* problem implemented in Objective-CP (Figure 5.9).

The implementation follows the description of the model given in section 1.2.3 very closely. The model itself is created on lines 1-24. The model is then *concretized* into a CP program on line 26. Finally, a simple custom search is defined on lines 27-31. Notice that even though the custom search is applied to a CP *program*, the implementation of the search

refers to variables in the original high-level formulation of the model. Hence, the underlying CP representation of the problem is completely opaque to the user.

Suppose, instead that the user wanted to run the *warehouse location* problem in a MIP solver. One option would be to remodel the problem similar to the IP formulation provided at the end of section 1.4.3. The user could then change the *concretization* in line 26 to instead use a MIP *program*. Although this approach would work in Objective-CP, and indeed would be the required approach in other environments including Comet, it fails to take advantage of the nice high-level modeling capabilities of Objective-CP. The better alternative would be to model the problem as in Figure 5.9, but instead add an additional line of code to apply a linear operator to the model before dropping it into the MIP program. The approach replaces lines 26-31 with the few lines in Figure 5.10. The MIP implementation is not using a custom search and instead relies on the default search of the underlying MIP solver (in this case Gurobi).

5.2.3 Search in Objective-CP

Search in Objective-CP falls into two categories, *blackbox search* and *custom search*. *Blackbox search* refers to an existing search procedure provided by the Objective-CP library that can be readily used with any model. *Blackbox searches* are the most widely used form of search as they employ well-studied *heuristics* that are known to perform well on a broad range of problems. Furthermore, *blackbox searches* are easy to deploy for users that are disinclined to write their own search and can readily be swapped around and experimented with. On many *optimization* platforms, *blackbox search* is the norm as very few platforms offer robust support for custom search. Below are some of the *blackbox searches* provided by Objective-CP:

First Fail is a very simple *variable ordering heuristic* that is among the most commonly used *search heuristics* for CP solvers. The strategy of this heuristic is to limit the size of the search space by detecting infeasible assignments as early as possible. This can

```
1 id<ORModel> mdl = [ORFactory createModel];
2 ORInt fixed = 30;
3 ORInt maxCost = 100;
4 id<ORIntRange> Stores     = RANGE(mdl,0,9);
5 id<ORIntRange> Warehouses = RANGE(mdl,0,4);
6 id<ORIntArray> cap = [ORFactory intArray:mdl array: @[@1,@4,@2,@1,@3]];
7
8 ORInt connection[10][5] = { ... };
9 ORInt* conn = (ORInt*)connection;
10
11 id<ORIntVarArray> cost = [ORFactory intVarArray: mdl range:Stores domain: RANGE(mdl,0,maxCost)];
12 id<ORIntVarArray> supp = [ORFactory intVarArray: mdl range:Stores domain: Warehouses];
13 id<ORIntVarArray> open = [ORFactory intVarArray: mdl range:Warehouses domain: RANGE(mdl,0,1)];
14
15 for(ORUInt i=Warehouses.low;i <= Warehouses.up;i++) {
16    [mdl add: [Sum(mdl,s, Stores, [supp[s] eq:@(i)]) leq:cap[i]]];
17 }
18 for(ORUInt i=Stores.low;i <= Stores.up; i++) {
19    id<ORIntArray> row = [ORFactory intArray:mdl range:Warehouses with:^ORInt(ORInt j) {
20        return conn[i*[Warehouses size]+j];}];
21    [mdl add: [[open elt:supp[i]] eq:@1]];
22    [mdl add: [cost[i] eq:[row elt:supp[i]]]];
23 }
24 [mdl minimize: [Sum(mdl,s, Stores, cost[s]) plus: Sum(mdl,w, Warehouses, [open[w] mul:@(fixed)])
      ]];
25
26 id<CPProgram> cp = [ORFactory createCPProgram:mdl];
27 [cp solve: ^{
28    [cp labelArray:cost orderedBy:^ORDouble(ORInt i) { return [cp domsize:cost[i]];}];
29    [cp labelArray:supp orderedBy:^ORDouble(ORInt i) { return [cp domsize:supp[i]];}];
30    [cp labelArray:open orderedBy:^ORDouble(ORInt i) { return [cp domsize:open[i]];}];
31 }];
```

Figure 5.9: *warehouse location* problem in Objective-CP

```
1 id<ORModel> lm = [ORFactory linearizeModel: mdl];
2 id<MIPProgram> p = [ORFactory createMIPProgram: mdl];
3 [p solve];
```

Figure 5.10: *warehouse location* problem transformed into MIP in Objective-CP

be achieved by branching on variables predicted to be the most likely to cause failures. In practice, this prediction is often kept simple and the solver branches on a variable with the smallest domain size.

Weighted Degree (see [15]) is a variable ordering heuristic which weights variable selection not only on domain size, but also on how likely the variable is to appear in a failed constraint. A counter (initialized to 1) is maintained for each constraint and incremented each time the constraint triggers a failure. The weight, $\alpha_{wdeg}(x)$, of a variable is computed over a set of constraints C_x (a subset of the set of constraints C):

$$C_x = \{c \in C \mid x \in vars(c) \wedge \exists \text{ unbound } y \in vars(c)\}$$

$$\alpha_{wdeg}(x) = \sum_{c \in C_x} weight(c)$$

The heuristic then orders variable branching by choosing a variable x with the smallest ratio $\frac{|D(x)|}{\alpha_{wdeg}(x)}$.

Impact-Based Search (see [74]) is a more sophisticated *heuristic* which is driven by estimations of impact (search space reduction) by the assignment of a variable $x = v$. A branch-and-bound tree search in CP progresses by branching on variables ($x = v$) and then performing propagation, which has the effect (typically) of shrinking the domains of other variables. At a node k, this process can be viewed as transforming a problem P_{k-1} into a smaller problem P_k. A measure, then, of the impact of the assignment $x = v$, can be captured as:

$$I(x \leftarrow v) = 1 - \frac{\mathcal{S}(P_k)}{\mathcal{S}(P_{k-1})}$$

where the function $\mathcal{S}(P)$ applied to a problem P gives an upper bound on the size of the search space by multiplying the sizes of the variable domains. A running estimate of the impact of assigning $x = v$ over a set of search tree nodes is maintained:

$$\bar{I}_k(x \leftarrow v) = \frac{\bar{I}_{k-1}(x \leftarrow v) \cdot (\alpha - 1) + I(x \leftarrow v)}{\alpha}$$

Where α is a search parameter allowing for weighting the relative value of past impacts versus present impact. Impacts at the root node are computed by simulating all possible variable-value assignments. Hence, IBS can have a large overhead on startup. The overall impact of a variable, can be estimated as $\sum_{v \in D(x)} 1 - \bar{I}(x \leftarrow v)$. Given these estimates, IBS performs a variable and value ordering heuristic during the search. A variable with the maximum impact is selected for branching and assigned to a value

with minimal impact.

Activity-Based Search (see [56]) is similar to Impact-Based search in that it maintains estimates of the 'activity' of variable domains following a branching decision which is used for a variable-ordering (and optionally value-ordering) heuristic. ABS computes the set of variables $X' \subseteq X$ (where X is the set of all variables) that have seen a reduction in their domain following branching and propagation. The variable activity is then defined as:

$$A(x) = A(x) \cdot \gamma, \quad \forall x \in X, \quad s.t. \, |D(x)| > 1$$
$$A(x) = A(x) + 1, \quad \forall x \in X'$$

The parameter $\gamma \in [0, 1]$ represents a decay rate allowing previous activity to age. The activity of an assignment $x = v$ at a node k is defined as $A_k(x = v) = |X'|$ and a running estimation of an assignment's activity can be tracked in a similar manner to IBS:

$$\bar{A}_1(x = v) = \frac{\bar{A}_0(x = v) \cdot (\alpha - 1) + A(x = v)}{\alpha}$$

Activities are initialized by probing on startup and a variable and value ordering heuristic can be defined in a manner similar to IBS.

Objective-CP features the same search capabilities as Comet built on a similar architecture of *continuations* and *search controllers* [93]. The major advantage in Objective-CP is that *search* is generic, that is, it is written in terms of a high-level abstract model and isn't dependent on the type of the underlying program.

5.3 Automating Hybrids: Runnables and Model Combinators

This section provides a new theoretical framework for building up arbitrarily sophisticated hybrid solvers from building blocks called *runnables*. This work builds on the concept of *model operators* introduced in *CML*, but provides new rigorous abstractions for capturing the semantics of a solver. In this way, the concept of *model operators* can be reconstructed and expanded as *model combinators*, operators capable of leveraging semantics in solver composition as well as error checking.

5.3.1 Models and Meta Data

Model transformations impose a natural relationship hierarchy among models. As a preliminary to defining sound model *combinators* [47], we first define an ordering over models as, for instance, it does little good to verify that two models are capable of exchanging upper bounds if the models are unrelated.

Definition 12 (Solution Set) *A solution for a model $M = \langle X, C, O\rangle$ is an assignment of all variables in X satisfying C. The set of solutions of model M is denoted by $Sol(M)$.*

Definition 13 (Projection of Solution Sets) *Consider a model $M = \langle X, C, O\rangle$ along with a solution s and $X' \subseteq X$. Then, $Sol|_{X'}(s)$ and $Sol|_{X'}(M)$ denotes the projection of solution s and the solution set of M on the variables in X', respectively.*

We now formalize the concept of relaxation, tightening, and equivalence of transformed models. Note that the model M' always has at least the same variables as M since M' is obtained through a transformation of M. Tightenings are only obtained by adding constraints, while relaxations can be obtained by adding variables in constraints or removing constraints.

Definition 14 (Relaxations and Tightenings of Satisfaction Problems) *Let $M = \langle X, C\rangle$ and τ be a transformation. The model $\tau(M) = M' = \langle X', C'\rangle$ is a relaxation of M, denoted by $M' \triangle M$, iff $Sol(M) \subseteq Sol|_X(M')$. It is a tightening, denoted by $M' \nabla M$, iff $Sol|_X(M') \subseteq Sol(M)$. M and M' are equivalent, denoted by $M \equiv M'$, if M' is both a relaxation and a tightening of M.*

Definition 15 (Relaxations and Tightenings of Minimization Problems) *Let $M = \langle X, C, O\rangle$ and and τ be a transformation. The model $\tau(M) = M' = \langle X', C', O'\rangle$ is a relaxation of M, denoted $M' \triangle M$, iff $\langle X', C'\rangle \triangle \langle X, C\rangle$ and*

$$\forall s \in Sol(M), s' \in Sol(M') : Sol|_X(s') = s \;\Rightarrow\; O'(s') \leq O(s).$$

M' is a tightening of M, denoted by $M' \nabla M$, if $\langle X', C'\rangle \nabla \langle X, C\rangle$ and

$$\forall s \in Sol(M), s' \in Sol(M') : Sol|_X(s') = s \;\Rightarrow\; O'(s') \geq O(s).$$

M and M' are equivalent, denoted by $M \equiv M'$, if M' is both a relaxation and a tightening of M.

The definitions of these concepts are transitive, reflexive and, for equivalences, commutative. We use $\triangle^*$ (resp. ∇^*) to denote the reflexive and transitive closure of $\triangle$ (resp. ∇). We use $\equiv^*$ to denote the reflexive, commutative, and transitive closure of $\equiv$. Combinators use these relations to enforce pre-conditions and post-conditions on their models.

5.3.2 Runnables and Model Signatures

This section introduces the concept of a *runnable*. Informaly, a *runnable* can be thought of as a producer/consumer process that uses a *program* to solve an optimization problem, consuming from a number of *input pipes*, and producing into a number of *output pipes* (see

Figure 5.11). The pipes deal with runnable products that are concepts such as solutions and bounds, as well as streams or sets of these products. A runnable is associated with a signature that specifies its inputs and outputs, i.e., the products that it consumes and produces. The implementation creates pipes for each of these inputs and outputs. If a runnable is executed directly, its input and output pipes are not used; the runnable pipes are only useful when it is combined with other runnables through *combinators* (see Figure 5.12). Note that stream pipes consume or produce products during the lifetime of a runnable; this is the case when exchanging solutions and bounds during the search.

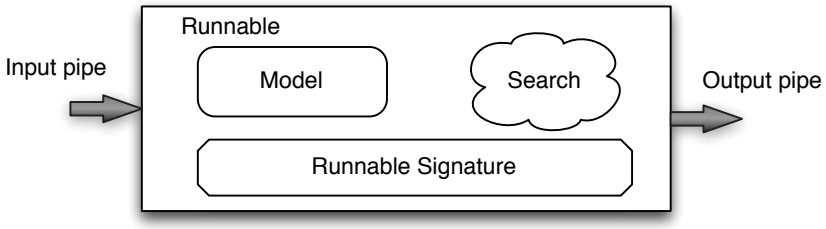

Figure 5.11: A runnable for solving a *process*

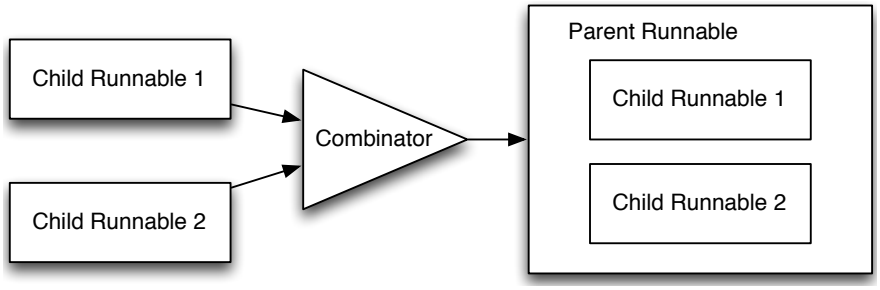

Figure 5.12: A composite from a combinator.

Definition 16 (Runnable Products) *A runnable product is specified by the grammar*

⟨*runnable product*⟩ *::=* ⟨*basic product*⟩ | `[`⟨*basic product*⟩`]` | {⟨*basic product*⟩}

⟨*basic product*⟩ *::=* `UBD` | `LBD` | `COL` | `CST` | `SOL`

where the basic products `UBD`, `LBD`, `COL`, `CST`, `SOL` *represent upper bounds, lower bounds, columns, constraints and solutions,* `[p]` *represents a stream of products of type* `p`*, and* {`p`} *a set of products of type* `p`*.*

Definition 17 (Runnable Signature) *A runnable signature is a pair $S = \langle I, O \rangle$, where I is a set of input runnable products and O is a set of output runnable products.*

Definition 18 (Runnable) *A runnable is a pair $R = \langle P, S \rangle$, where P is an optimization program and S is a runnable signature.*

We often abuse language and talk about the model of a runnable to denote the model of its program.

Definition 19 (Pipes of a Runnable) *Let R be a runnable $\langle P, \langle I, O \rangle \rangle$. R provides the set of input pipes $\{in(p, R) \mid p \in I\}$ and the set of output pipes $\{out(p, R) \mid p \in O\}$.*

The implementation provides a number of primitive runnables. They can be created from a model M, a flattening, and a concretization. For instance, the `CPRunnable` has a program $\langle \texttt{flatten}(M), \gamma_{CP} \rangle$ and a predefined signature.

5.3.3 Model Combinators

This section describes model combinators. We restrict our attention to binary operators for simplicity but it is easy to generalize the results for non-binary combinators. A model combinator $R = C(R_1, R_2)$ combines two runnables R_1 and R_2 to produce another runnable. The combinator requires some properties from its runnables, establishes the links between the pipes of its runnables and its own, and specifies how its model relates to the models of its runnables. Figure 5.13 illustrates the piping intuitively. More precisely, the specification of a model combinator consists of several parts:

1. A precondition that specifies the required relationships between the runnable models and the existence of some input/output products.

2. A set of piping rules for linking the input pipes of the combinators to the input pipes of its runnables.

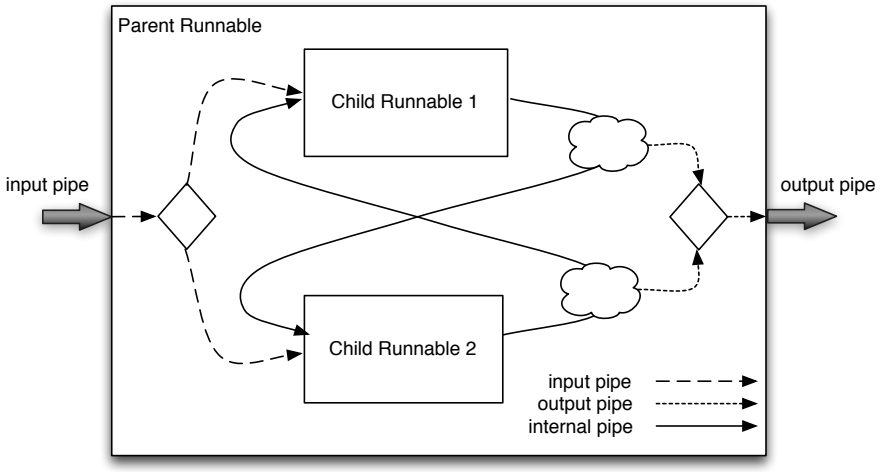

Figure 5.13: A Composite Runnable and Its Input, Output and Internal Pipes.

3. A set of piping rules for linking the output pipes of its runnables to its output pipes.

4. A set of piping rules for linking the pipe of the runnables.

5. A relationship between the model of the combinator and the model of the runnables.

A piping rule is an expression of the form $\pi_1 \rightarrow \pi_2$ which specifies that pipe π_1 produces products that are consumed by pipe π_2. For instance, the rule

$$in(\mathrm{SOL},R) \rightarrow in(\mathrm{SOL},R_1)$$

specifies that the input pipe for solutions in R produces solutions that are consumed by the input pipe for solution in R_1. If p is a product, an input pipe rule is of the form

$$in(\mathrm{p},\ \mathrm{R}) \rightarrow in(\mathrm{p},\mathrm{R}_i)$$

an output pipe rule is of the form

$$out(\mathrm{p},\mathrm{R}_i) \rightarrow out(\mathrm{p},\mathrm{R})$$

and an internal pipe rule is of the form

$$out(\mathrm{p},\mathrm{R}_i) \rightarrow in(\mathrm{p},\mathrm{R}_j)$$

It is also useful to allow output piping rules with no antecedent, i.e.,

$\rightarrow$ *out*(p,R)

for situations where the combinator products are not directly taken from the runnables but computed by the combinator itself.

These piping rules have two main purposes: To establish the plumbing inside the combinators and to synthesize the signature of the combinator. It is important to state that the combinator does not have a static signature. Rather the most general signature is synthesized based on the functionalities of its runnables.

Definition 20 (Combinator Specification) *Let R_1 and R_2 be two runnables with signatures $S_i = \langle I_i, O_i \rangle$ and models M_i ($1 \leq i \leq 2$). The specification of a combinator $C(R_1, R_2)$ is a tuple $\langle \mathcal{P}, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{M} \rangle$, where $\mathcal{P}$ is a precondition on M_i, I_i, and O_i, $\mathcal{I}$, $\mathcal{O}$, $\mathcal{E}$ are sets of input, output, and internal piping rules, and $\mathcal{M}$ specifies the model relationship.*

Obviously, the combinator does not have a model on its own: It combines, sometimes in complex ways, the models of its runnable. Hence, the model relationship specifies the semantics of its products, such as its solutions, its bounds, and streams thereof. For instance, a model relationship $R \triangle R_1$ specifies that the (implicitly defined) combinator model is a relaxation of the model of R_1. The new information is propagated through the transitive closures of $\triangle$ in order to verify preconditions involving R in subsequent combinations. We are now ready to synthesize the combinator signature.

Definition 21 (Combinator Signature) *Let R_1 and R_2 be two runnables with signatures $S_i = \langle I_i, O_i \rangle$ and a combinator $R = C(R_1, R_2)$ with specification $\langle \mathcal{P}, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{M} \rangle$. The signature of R is $\langle I, O \rangle$ where*

$$
\begin{aligned}
I &= \{\, p \mid in(p, R) \rightarrow in(p, R_i) \in \mathcal{I} \wedge p \in I_i \wedge 1 \leq i \leq 2 \} \\
O &= \{\, p \mid out(p, R_i) \rightarrow out(p, R) \in \mathcal{O} \wedge p \in O_i \wedge 1 \leq i \leq 2 \}.
\end{aligned}
$$

Observe once again that the definition of input/output is dynamic: The piping rules defines what is possible and the actual input/output definitions are derived from the actual input and

output functionalities of the combined runnables. If a runnable does not provide a certain product (e.g., streams of lower bounds), this product is not synthesized in the signature, even if a piping rule was specified. We are now ready to present some combinators.

5.4 Examples of Model Combinators

In the following subsections, concrete examples of combinators will be described in detail. To ease the description of these combinators, assume that the following two runnables are defined:

$$R_1 = \langle P_1 = \langle M_1, S_1 \rangle, T_1 = \langle I_1, O_1, \mathcal{T}_1 \rangle \rangle$$
$$R_2 = \langle P_2 = \langle M_2, S_2 \rangle, T_2 = \langle I_2, O_2, \mathcal{T}_2 \rangle \rangle$$

Each subsection will provided a textual description of the combinator as well as a formal semantic description.

5.4.1 Sequential Combinator

Description: An upper bound passing sequential combinator is the simplest combinator to define. A sequential combinator $R = R_1 \triangleright R_2$ uses R_1 to compute an upper bound which is then passed as an input to R_2 . This combinator (see Figure 5.14) is often used in practice when a heuristic search first finds a high-quality upper bound which is then used to seed a systematic search. The combinator specification is as follows:

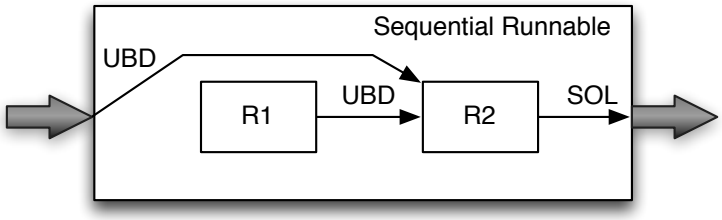

Figure 5.14: Runnable produced by sequential combinator.

Semantics: The precondition requires that M_1 be a tightening of M_2 to ensure that the upper

bound of M_1 is indeed an upper bound for M_2. The input piping rule allows the upper bound of R to be consumed by R_2; It cannot be passed to R_1 since R_1 is a tightening of M_2. The output piping rule allows the solution of R_2 to be produced as a solution to R. The internal piping rule specifies that the upper bound produced by R_1 can be consumed by R_2. The model relationship specifies that the resulting combinator is equivalent to R_2.

The signature that is synthesized here is trivial, since the piping rules are only concerned with required inputs and outputs. It will simply be $\langle\{UBD\},\{SOL\}\rangle$. If the output piping rule

$$out(LBD, R_2) \rightarrow out(LBD, R)$$

had been present, and LBD would belonged to O_2, the synthesized signature would have been $\langle\{UBD\},\{SOL, LBD\}\rangle$.

$$R = R_1 \triangleright R_2$$

$\mathcal{P}_{\triangleright} =$	$R_1 \nabla^* R_2 \wedge UBD \in O_1 \wedge UBD \in I_2 \wedge SOL \in O_2$
$\mathcal{I}_{\triangleright} =$	$\{in(UBD, R) \rightarrow in(UBD, R_2)\}$
$\mathcal{O}_{\triangleright} =$	$\{out(SOL, R_2) \rightarrow out(SOL, R)\}$
$\mathcal{E}_{\triangleright} =$	$\{out(UBD, R_1) \rightarrow in(UBD, R_2)\}$
$\mathcal{M}_{\triangleright} =$	$R \equiv R_2$

Example: Figure 5.15 demonstrates a typical use case of this combinator in OBJECTIVE-CP. The example runs a *jobshop problem* (see section 1.2.4), but the procedure is similar for any problem. The model is defined on line 1. Lines 3-10 create a CP runnable with a custom LNS search. Line 6 specifies that the search should run for only 5 seconds. Lines 12-18 create a complete CP solver using the global slack heuristic . Line 20 calls the sequential combinator to combine the two runnables into a single sequential runnable which is then run on line 21. Running this hybrid on standard jobshop instances can provide an order of magnitude speedup (on `orb10` for example) over the complete CP search alone.

```
1 id<ORModel> m = /* Model jobshop problem */
2
3 id<ORRunnable> r0 = [ORFactory CPRunnable: m
4 solve:^(id<CPProgram,CPScheduler> lns) {
5       [lns solve: ^{
6           [lns limitTime: 1000L * 5 in:^{
7                   /* jobshop custom LNS */
8           }];
9       }];
10 }];
11
12 id<ORRunnable> r1 = [ORFactory CPRunnable: m
13 solve:^(id<CPProgram,CPScheduler> cp) {
14     [cp solve: ^{
15                /* jobshop complete search
16                   using global slack heuristic */
17     }];
18 }];
19
20 id<ORRunnable> r = [ORFactory composeSequential: r0 with: r1];
21 [r run];
```

Figure 5.15: *Sequential combinator* in OBJECTIVE-CP running an LNS search for 5 seconds to get a bound for a complete CP search.

5.4.2 Complete Parallel Combinator

Description: A parallel solution passing combinator between two *runnables* with models in the same equivalence class (see figure 5.16). That is, this combinator produces a runnable which runs its children concurrently and sets up communication channels for real-time exchange of solutions or bounds.

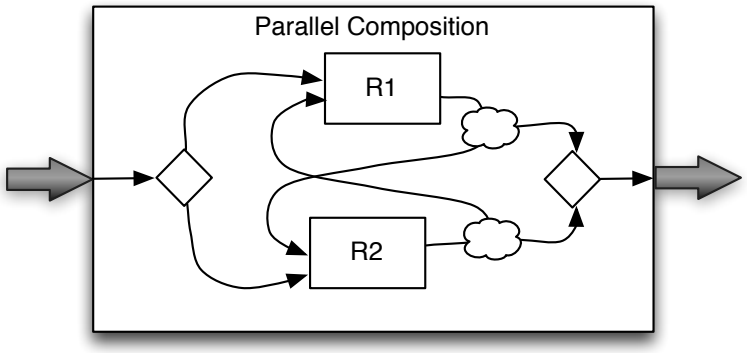

Figure 5.16: Runnable produced by complete parallel combinator.

Semantics: The precondition $\mathcal{P}_{||}$ ensures that the two runnables are equivalent and the input and output of both runnables include a stream of solutions. The piping rules are very explicit this time and allow for the exchanges of upper and lower bounds as well. In particular, if the

runnables provide lower bounds, the implementation will ensure that the internal piping provides that functionality. Similarly, the input and output piping will synthesize streams of upper and lower bounds if the combined runnables provide these products.

Observe that this combinator can be used for composing three runnables: It suffices to use $(R_1 \parallel R_2) \parallel R_3$, since the parallel combinator will satisfy its own precondition. Also, Figure 5.16 shows the flow of solutions within this parallel runnable using black arrows, The small clouds waiting at the outputs of the child runnables represents small blocks of code used by the parent to intercept output solutions coming from the children.

$$R = R_1 \parallel R_2$$

$\mathcal{P}_{\parallel} =$	$R_1 \equiv^* R_2 \wedge [SOL] \in I_1 \wedge [SOL] \in O_1 \wedge [SOL] \in I_2 \wedge [SOL] \in O_2$
$\mathcal{I}_{\parallel} =$	$\{in([SOL], R) \rightarrow in([SOL], R_1), in([SOL], R) \rightarrow in([SOL], R_2),$ $in([UBD], R) \rightarrow in([UBD], R_1), in([UBD], R) \rightarrow in([UBD], R_2),$ $in([LBD], R) \rightarrow in([LBD], R_1), in([LBD], R) \rightarrow in([LBD], R_2)\}$
$\mathcal{O}_{\parallel} =$	$\{out([SOL], R_1) \rightarrow out([SOL], R), out([SOL], R_2) \rightarrow out([SOL], R),$ $out([UBD], R_1) \rightarrow out([UBD], R), out([UBD], R_2) \rightarrow out([UBD], R),$ $out([LBD], R_1) \rightarrow in([LBD], R), out([LBD], R_2) \rightarrow out([LBD], R)\}$
$\mathcal{E}_{\parallel} =$	$\{out([SOL], R_1) \rightarrow in([SOL], R_2), out([SOL], R_2) \rightarrow in([SOL], R_1),$ $out([UBD], R_1) \rightarrow in([UBD], R_2), out([UBD], R_2) \rightarrow in([UBD], R_1),$ $out([LBD], R_1) \rightarrow in([LBD], R_2), out([LBD], R_2) \rightarrow in([LBD], R_1)\}$
$\mathcal{M}_{\parallel} =$	$R \equiv R_1$

Example: Figure 5.17 demonstrates a complete parallel combinator generating an LNS/MIP hybrid for the *jobshop problem* (see section 1.2.4). Rather than using a complete CP solver as was done for the *sequential combinator*, this example will use a MIP solver. The model is defined on line 1. Lines 3-8 create a CP runnable with a custom LNS search. Note that, unlike the *sequential combinator*, no time limit is set for the LNS search. Lines 10-11 linearize the jobshop model,

producing a new linear abstract model which is semantically equivalent to the global constraint formulation. Line 12 creates a MIP runnable from the linearized model and then line 13 calls the complete parallel combinator to combine the two runnables.

```
1 id<ORModel> m = /* Model jobshop problem */
2
3 id<ORRunnable> r0 = [ORFactory CPRunnable: m
4 solve:^(id<CPProgram,CPScheduler> lns) {
5       [lns solve: ^{
6          /* jobshop custom LNS */
7       }];
8 }];
9
10 id<ORModel> lm = [ORFactory linearizeSchedulingModel: m
11         encoding: MIPSchedDisjunctive];
12 id<ORRunnable> r1 = [ORFactory MIPRunnable: lm];
13 id<ORRunnable> r = [ORFactory composeCompleteParallel: r0 with: r1];
14 [r run];
```

Figure 5.17: *Parallel combinator* in OBJECTIVE-CP running an LNS in CP concurrently with a complete MIP.

5.4.3 Relaxed Parallel Combinator

Description: The Relaxed Parallel Combinator, like the *complete parallel combinator*, runs two models concurrently. In this case, however, M_2 is a relaxation of M_1 and M_2 passes lower bounds to M_1.

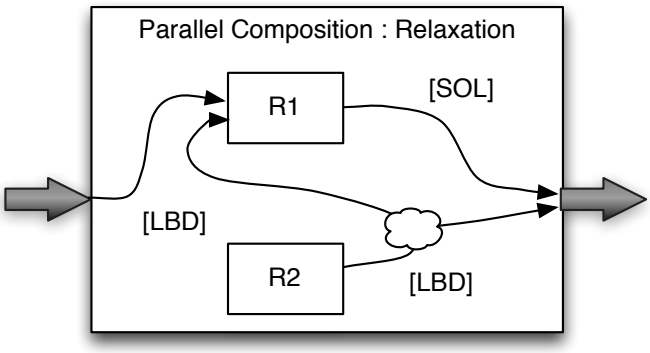

Figure 5.18: Runnable produced by relaxed parallel combinator.

Semantics: The precondition $\mathcal{P}_{\dashv}$ ensures that R_2 is a relaxation of R_1, R_1 produces a stream of solutions, and R_2 produce lower bounds. The input piping rules $\mathcal{I}_{\dashv}$ states that streams of lower bounds, upper bounds or solutions consumed by $R_{\dashv}$ can be consumed by R_1. The output piping

rules $\mathcal{O}_{\dashv}$ state that $R_{\dashv}$ produces the output streams produced by R_1. The internal piping rules ensure that the stream of lower bounds produced by R_2 can be consumed by R_1. The combinator then produces a model equivalent to R_1. Figure 5.18 illustrates the flow of runnable products through this runnable assuming the children meet only the minimum preconditions for simplicity.

$$R = R_1 \dashv R_2$$

$\mathcal{P}_{\dashv} =$	$R_2 \triangle^* R_1 \wedge [SOL] \in O_1 \wedge [LBD] \in I_1 \wedge [LBD] \in O_2$
$\mathcal{I}_{\dashv} =$	$\{in([LBD], R) \rightarrow in([LBD], R_1), in([UBD], R) \rightarrow in([UBD], R_1),$ $in([SOL], R) \rightarrow in([SOL], R_1)\}$
$\mathcal{O}_{\dashv} =$	$\{out([SOL], R_1) \rightarrow out([SOL], R), out([UBD], R_1) \rightarrow out([UBD], R),$ $out([LBD], R_1) \rightarrow in([LBD], R)\}$
$\mathcal{E}_{\dashv} =$	$\{out([LBD], R_2) \rightarrow in([LBD], R_1)\}$
$\mathcal{M}_{\dashv} =$	$R \equiv R_1$

5.4.4 Column Generation Combinator

Description: Automating column-generation solvers has been done previously in systems such as CML [38] and the G12 Project [69], but the use of runnables allows for a cleaner expression of the semantics as well as a much more compositional interface. The column-generation combines a master problem and a slave problem. The master runnable consumes columns and generates solutions, while the slave runnable consumes solutions and produces columns.

Semantics: Column Generation does not impose any runtime restrictions on the slave closure as the necessary conditions are checked statically by the compiler. The slave closure will typically make use of a runnable internally, but it need not do so, producing a column will suffice. The use of a closure for running the slave problem is quite useful as the closure can be run repeatedly to generate new columns while capturing the latest solution from the master at each iteration automatically.

The above precondition $\mathcal{P}_{\bowtie}$ states that the *master* accepts a stream of columns and generates

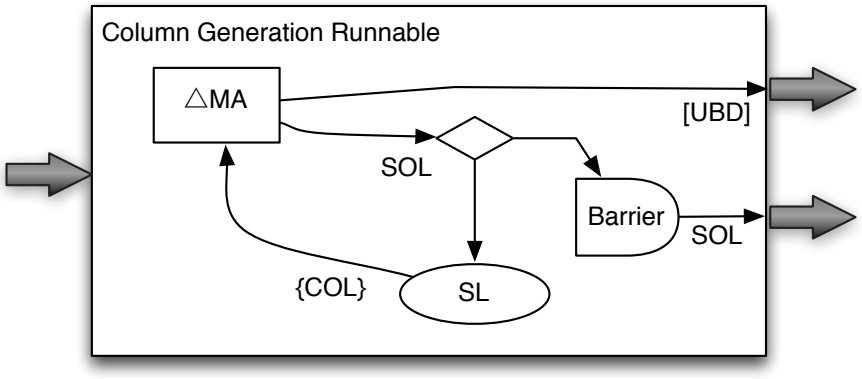

Figure 5.19: Runnable produced column generation combinator.

a solution. The output relations $\mathcal{O}_{\bowtie}$ states that the output pipe of $R_{\bowtie}$ produces a stream of upper bounds as well as a solution. Finally, the internal pipe relations ensure that the master outputs a solution to the slave closure and that the slave produces a set of columns as input to the master.

An implementation of the combinator copies the master runnable, before starting the column-generation process, in order to allow the master runnable to be reused in other combinators. As a result, the combinator does not use the output of its runnables but generates products on its own. Note that the column-generation combinator is very general: It does not impose how the slave process uses the solution (though the dual values are captured in the solution). As a consequence, it can implement a traditional column-generation algorithm or use a heuristic approach to generate columns based on the problem structure.

$$R = MA \bowtie SL \text{ with } MA = \langle I_m, O_m \rangle \text{ and } SL = \langle I_s, O_s \rangle$$

$\mathcal{P}_{\bowtie} =$	$COL \subseteq I_m \wedge SOL \subseteq O_m \wedge SOL \subseteq I_s \wedge COL \subseteq O_s$
$\mathcal{I}_{\bowtie} =$	$\{\}$
$\mathcal{O}_{\bowtie} =$	$\{\rightarrow out(SOL, R), \rightarrow out([UBD], R)\}$
$\mathcal{E}_{\bowtie} =$	$\{out(\{COL\}, SL) \rightarrow in(\{COL\}, MA), out(SOL, MA) \rightarrow in(SOL, SL)\}$
$\mathcal{M}_{\bowtie} =$	$R \triangle MA$

Example: Our example for column generation will again be *cutting stock*. For a detailed overview of solving this problem with column generation, refer to section 3.3. Lines 1-16 define a master

problem and create an LP runnable capable of solving it. Line 18 calls the column generation combinator, taking the *master* runnable and a closure that defines the *slave*. The closure spans lines 19-39 and must either produce and return a column or return *nil* to terminate the method. Within the closure, the *slave* is modeled on lines 21-27 and solved on lines 29-30. Lines 32-39 produce a column from the variable array *use* if the *reduced cost* is negative or return nil otherwise. The column generation runnable is executed on line 41.

```
1 // Model master
2 id<ORModel> master = [ORFactory createModel];
3 ORInt width = ...;
4 ORInt shelfCount = ...;
5 id<ORIntRange> shelves = RANGE(master,0,shelfCount - 1);
6 id<ORIntArray> shelf   = [ORFactory intArray: master  array: ...];
7 id<ORIntArray> demand  = [ORFactory intArray: master array: ...]];
8 id<ORIntMatrix> columns = /* Initial columns */
9 id<ORRealVarArray> cut = [ORFactory realVarArray: master
10        range: shelves low:0 up:[demand max]];
11 for(ORInt i = [shelves low]; i <= [shelves up]; i++) {
12    [master add: [Sum(master, j, shelves, [cut[j] mul: @([columns[j][i]])])
13        geq: @(demand[i])]];
14 }
15 [master minimize: Sum(master, i, shelves, cut[i])];
16 id<ORRunnable> lp = [ORFactory LPRunnable: master];
17 // Create column generation runnable with closure
18 id<ORRunnable> r = [ORFactory columnGeneration: lp
19        slave:^id<ORDoubleArray>(id<ORDoubleArray> cost) {
20    // Model slave problem
21    id<ORModel> slave = [ORFactory createModel];
22    id<ORIntVarArray> use =
23        [ORFactory intVarArray: slave range: shelves domain: shelves];
24    [slave minimize: [@(1) sub:
25        Sum(slave, i, shelves, [use[i] mul: @(cost[i])])]];
26    [slave add: [Sum(slave, i, shelves, [@(shelf[i]) mul: use[i]])
27        leq: @(width)]];
28    // Solve slave
29    id<MIPProgram> ip = [ORFactory createMIPProgram: slave];
30    [ip solve];
31    // Create column
32    id<ORSolution> sol = [[ip solutionPool] best];
33    ORDouble reducedCost = [[sol objectiveValue] doubleValue];
34    id<ORDoubleArray> col = nil;
35    if(reducedCost < -0.00001) {
36        col = [ORFactory doubleArray: slave range: [use range] with:
37                        ^ORDouble(ORInt i) { return [sol intValue: use[i]]; }];
38    }
39    return col;
40 }];
41 [r run];
```

Figure 5.20: *Column generation combinator* in OBJECTIVE-CP running the cutting stock problem.

5.4.5 Logic-Based Bender's Combinator

Description: Logic-Based Benders was not supported in CML but this section highlights that it is in fact the dual of the column-generation combinator and is easily supported in Objective-CP. For a more detailed description of this method, see section 3.4. Informally speaking, in its simplest form, a Benders decomposition features a master that relaxes some of the constraints of an original model and a slave that checks if the solution produces by this master are feasible for the relaxed constraints. If these constraints are infeasible, the slave generates new constraints (cuts) that are added to the master. The process is repeated until a feasible (and optimal) solution is found. Once again, the combinator receives a master and a slave runnable. The master runnable is copied and the combinator implementation keeps adding constraints to the master until an optimal solution is found. The slave receives the solutions to the master and generates new constraints. The combinator produces a stream of lower bounds and a final solution. The model specification closely mirrors the combinator for column generation, with upper bounds being replaced by lower bounds. Moreover, the combinator is now a tightening of the master program since the Benders decomposition adds new constraints. Figure 5.21 illustrates the combinator.

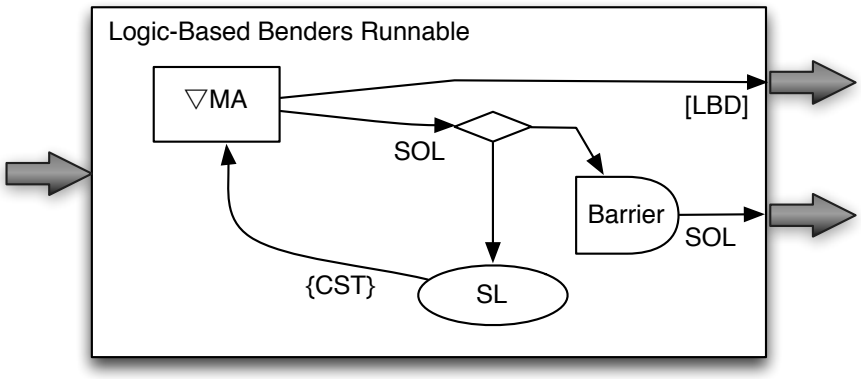

Figure 5.21: Runnable produced by the logic-based benders combinator.

Semantics: The precondition $\mathcal{P}_{\otimes}$ checks that the *master* accepts a pool of constraints and generates a solution. The input relations $\mathcal{I}_{\otimes}$ states that the master can receive a stream of constraints. The output relations $\mathcal{O}_{\otimes}$ states that the output pipe of $R_{\otimes}$ produces a solution taken from the master. Finally, the internal pipe relations ensures that the master outputs a solution to the slave closure. The slave closure will use the solved master problem to generate and run a slave problem

before outputting a set of constraints (cuts) which will be injected into the master.

$R = MA \otimes SL$ with $MA = \langle I_m, O_m \rangle$ and $SL = \langle I_s, O_s \rangle$

$\mathcal{P}_{\otimes} =$	$CST \subseteq I_m \wedge SOL \subseteq O_m \wedge SOL \subseteq I_s \wedge CST \subseteq O_s$
$\mathcal{I}_{\otimes} =$	$\{\}$
$\mathcal{O}_{\otimes} =$	$\{\rightarrow out(SOL, R), \rightarrow out([LBD], R)\}$
$\mathcal{E}_{\otimes} =$	$\{out(\{CST\}, SL) \rightarrow in(\{CST\}, MA), out(SOL, MA) \rightarrow in(SOL, SL)\}$
$\mathcal{M}_{\otimes} =$	$R \nabla MA$

Example: Figure 5.22 shows an the outline of an implementation for the *warehouse location-allocation problem* in OBJECTIVE-CP. The modeling of the *master* and *slave* problems are omitted, but are identical to the models in section 3.4. Line 1 models the *master* problem and line 2 creates a MIP runnable. Line 3 calls the Logic-Based Benders combinator with the *master* runnable and a closure that runs the *slave* and returns a pool of cuts. The combinator will iterate between the *master* and *slave* problems until the closure returns an empty set of cuts. The closure spans lines 4-29 and iterates over the index of each open warehouse location (j). The Fit First Decreasing Heuristic is run on line 11 and is used to check if the bin packing subproblem needs to run on lines 16-17. If the pin packing problem runs and is infeasible, a cut is generated on lines 22-23. The hybrid runnable is executed on line 30.

```
id<ORModel> master = /* Model master problem */
id<ORRunnable> ip = [ORFactory MIPRunnable: master];
id<ORRunnable> benders = [ORFactory logicBenders: ip slave:
        ^id<ORConstraintSet>(id<ORSolution> sol) {
    id<ORConstraintSet> cuts = [ORFactory createConstraintSet];
    for (ORInt j = 0; j < n; j++) {
        id<ORIntSet> Ij = /* collect open stores for warehouse j */
        if([Ij size] == 0) continue; // warehouse not being used
        id<ORIntArray> dist = /* collect distances for Ij */;

        ORInt numVehFFD = FirstFitDecreasingHeuristic(l, dist);
        ORInt numVehj = [sol intValue: [numVeh at: j]];
        if(numVehFFD > numVehj) {
            // Run slave
            for(ORInt numVehBinPacking = numVehj; numVehBinPacking < numVehFFD; numVehBinPacking
                ++) {
                id<ORModel> slave = /* model binpacking problem */
                id<ORRunnable> r = [ORFactory CPRunnable: slave];
                [r run];
                id<ORSolution> best = [r bestSolution];
                if(best == nil) { // Infeasible
                    // Add Cut to pool
                    id<ORConstraint> c = /* model cut */;
                    [cuts addConstraint: c];
                }
            }
        }
        return cuts;
    }];
[benders run];
```

Figure 5.22: *Logic-Based Benders combinator* in OBJECTIVE-CP running the warehouse location-allocation problem.

5.5 Case Studies

This section presents benchmark results to assess the practicality of model combinators. The goal is not to give comprehensive results on a wide variety of benchmarks but to give evidence that this is a promising approach to ease the building of hybrid optimization algorithms.

The first benchmark is the Location-Allocation Problem implemented using the logical Benders approach presented in section 3.4 from [32]. It will allow us to compare the efficiency of an automated model with a hand-crafted implementation. The experiments feature 6 instances from the original paper, namely, the *uncorrelated* 20×10 (20 clients, 10 facilities) instances. The authors reported an average running time of 33 seconds for these instances. The combinator results are based on 20 runs of each instance and are given in Table 5.23(a). The Bender's runnable runs in about 39 seconds on average which is remarkably close to the results for the hand-written model. The experiment was carried out on a 2.13 Ghz Intel Core 2 Duo with 4 GB of RAM running Mac OS X (10.8) which is comparable to the machines in the original paper (Duo Core AMD 270 CPU, 4

Instance #	min	max	avg
1	19.69	21.23	20.23
2	22.53	26.66	23.82
3	11.26	13.30	11.92
4	5.81	7.61	6.37
5	94.81	110.13	99.31
6	67.28	79.73	70.91
overall avg			38.76

(a) Logic-Based Bender's

Inst	min	max	avg	min ‖	max ‖	avg ‖
8×20	2.2	2.5	2.3	1.8	2.3	2.1
8×30	6.1	6.5	6.3	2.5	4.3	3.5
8×40	7.9	16.2	9.7	4.5	9.5	5.7
9×20	72.0	89.8	75.3	48.6	62.3	55.5
9×30	27.6	30.2	28.3	21.8	27.9	24.1
9×40	155.2	174.5	165.6	72.6	89.5	80.2

(b) Assignment Problem

Figure 5.23: Benchmarks for Objective-CP runnables.

GB Ram, Red Hat Linux). Table 5.24 reports the results of an instrumentation of code to measure the time spent in the master, in the slave, and otherwise, considering that the remainder of time was attributed to the combinator. This is an overestimate of the true combinator cost as any other overhead is attributed to the combinator. The Master columns report the total time spent in the master. The % column report the fraction of the total that this represents. The same applies for the Slave and Combinator columns. For any row, the percentages add up to 100%. Overall, the combinator overhead never exceeds $\frac{1}{2}$% of the runtime and demonstrates that the approach is competitive. This should be contrasted with the brief combinator-based implementation which weighs in at 90 lines of Objective-C code (without data reading) to create the models and setup the Bender's combinator.

The second benchmark is a simple Assignment Problem (AP) in which we run a standard CP implementation [38] in parallel with a CP linear reformulation using the Complete Parallel Combinator. Note that there are better approaches to solving the AP, we only aim to show the benefit of using combinators to generate a parallel runnable. The linear reformulation is substantially slower (particularly as a CP model). The table gives the running time of solving the linear model alone and within the parallel runnable. Results are based on random instances with sizes $n \times m$ where n

	Master			Slave			Combinator		
Instance#	μ	σ	%	μ	σ	%	μ	σ	%
1	20.12	0.40	99.41	0.09	0.02	0.42	0.12	0.40	0.17
2	23.72	0.90	99.57	0.06	0.02	0.25	0.10	0.90	0.18
3	11.84	0.46	99.37	0.04	0.01	0.31	0.08	0.47	0.32
4	6.31	0.57	99.10	0.03	0.01	0.43	0.05	0.57	0.47
5	99.14	3.68	99.82	0.14	0.05	0.14	0.18	3.71	0.04
6	70.80	3.10	99.84	0.07	0.03	0.12	0.11	3.10	0.04

Figure 5.24: Time allocation between Master/Slave/Combinator.

is the number of agents/tasks and m is the maximum allowed cost (cost range $\in [1, m]$). Columns *min, max, avg* in Table 5.23(b) represent the minimum maximum and average running time (in secs) of the standalone linear CP problem, while *min* $\|$, *max* $\|$, *avg* $\|$ refer to the parallel runnable.

Chapter 6

Application: Parallel Portfolio Solvers

Portfolio solvers (introduced in section 3.5) have proven to be among the most effective hybrid techniques in recent years, attracting significant research attention as well as producing solvers capable of easily winning SAT solver competitions (SATzilla [101], for example). Portfolio solvers have typically been comprised of isolated solvers running in competition with one and other. There are several reasons for this, a major one being that building concurrent, cooperative solvers requires a great deal of effort. Part of the appeal of early portfolio solver approaches was that they could be thrown together relatively easily and yet often produced very good results. Furthermore, if a well-tuned solver can be found for a particular problem instance, it often can solve the problem with relative ease and does not benefit much from interaction with other ill-tuned solvers. Hence, most of the research effort around portfolio solvers has been focused on machine learning or clustering technologies for building portfolios of well-tuned solvers [46] [62] .

As portfolio approaches are applied to increasingly difficult problems and are now often comprised of high-quality solvers, the potential benefits of a cooperative strategy are substantial. The *parallel combinator* presented in chapter 5.4.2 eliminates the major barrier to this, allowing solvers to be combined in a cooperative fashion with ease. Furthermore, modern desktop and laptop systems are overwhelmingly parallel machines with 2-8 cores. Parallel tree search [88, 65, 78, 53, 75, 57] has been under investigation for two decades and is possibly the sole effort to exploit small and large scale parallelism. Despite the advent of parallel hardware and the absence of dominating combinatorial techniques, surprisingly little has been done to produce *robust* parallel algorithmic

combinatorial optimization techniques. The aim of this chapter is to illustrate the benefits associated with pursuing a cooperative portfolio approach and to suggest that parallel portfolio solvers may offer a compelling alternative to Parallel Tree Search for leveraging parallelism on modern hardware.

The problem domain this chapter considers is *jobshop scheduling*. For an overview of the problem, refer to section 1.2.4. Scheduling has long been considered a strength of Constraint Programming (CP) solvers. Recent work on Failure-Directed Search [98] demonstrates that CP continues to provide state-of-the-art results. Work done in the last few years, e.g., [50] also shows that modern Mixed-Integer Programming (MIP) solvers using standard encodings are now competitive with, and sometimes superior to, commonly used CP scheduling solvers. Figure 6.1 shows running times (in seconds) of CP and MIP solvers on standard jobshop benchmarks. Indeed, the figure shows superior results for CP on 5 instances and superior performance for MIP on 4 (with both timing out on one instance).

Such results suggest that composite techniques leveraging both technologies are in order. This chapter extends *model combinators* to the scheduling domain and demonstrates that, with just a few lines of code, a high level model can yield a multi-technology composite solver that can sometimes substantially outperform standalone MIP or CP solvers.

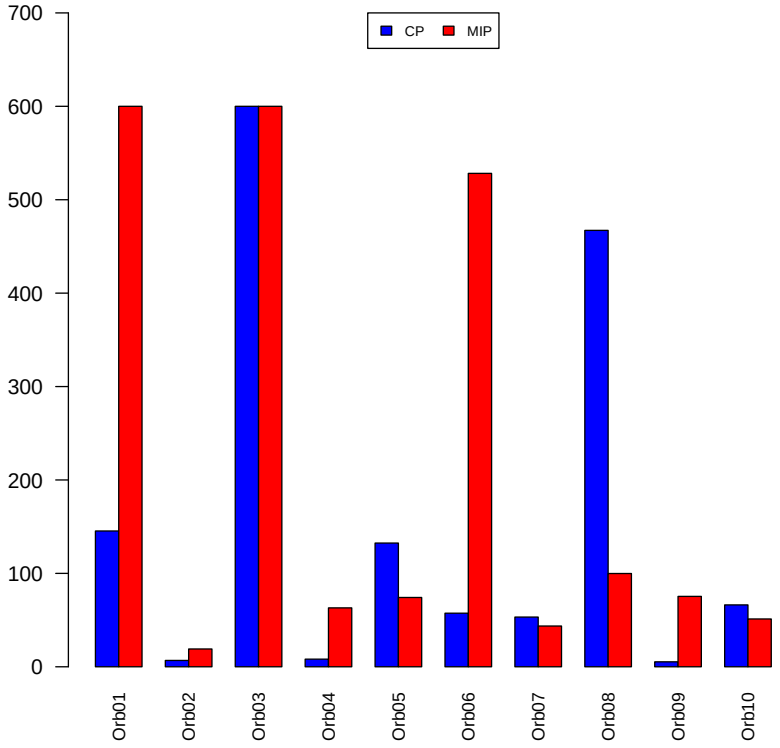

Figure 6.1: Running time in seconds of CP & MIP solvers running on standard instances of the jobshop scheduling problem. A timeout of 600s was used.

6.1 Composition of Scheduling Solvers

6.1.1 An Objective-CP Jobshop Model

With Objective-CP, models are containers capturing constraints that must be satisfied as well as a relevant objective function. Figure 6.2 illustrates the creation of a high-level declarative model. Line 1 creates a model m while lines 3-4 create ranges for the jobs J and machines M. Lines 5-6 create matrices holding the processing time of the activities as well the resource that any activity require. Line 8 creates a matrix $task$ holding all the activities. Line 9 creates a variable representing the makespan of the instance and line 10 creates an array of disjunctive resources (as many as M). Lines 12-22 start by stating the objective function and creating the constraints. The loops state the job precedence constraints, the fact that the makespan follows the end of each job, enforces the

```
1 id<ORModel> m = [ORFactory createModel];
2 // data setup ...
3 id<ORIntRange> J         = RANGE(m,0,nbJobs-1);
4 id<ORIntRange> M         = RANGE(m,0,nbMach-1);
5 id<ORIntMatrix> D        = [ORFactory intMatrix: m range: J : M];
6 id<ORIntMatrix> resource = [ORFactory intMatrix: m range: J : M];
7 // variables
8 id<ORTaskVarMatrix> task = [ORFactory tvMatrix:m range:J:M horizon:H duration:D];
9 id<ORIntVar> makespan    = [ORFactory intVar: m domain: RANGE(m,0,totalDur)];
10 id<ORTaskDisjunctiveArray> disjunctive = [ORFactory disjunctiveArray:m range: M];
11 // model
12 [m minimize: makespan];
13 for(ORInt i = J.low; i <= J.up; i++)
14    for(ORInt j = M.low; j < M.up; j++)
15       [m add: [[task at: i : j] precedes: [ task at: i : j+1]]];
16 for(ORInt i = J.low; i <= J.up; i++)
17    [m add: [[task at: i : Machines.up] isFinishedBy: makespan]];
18 for(ORInt i = J.low; i <= J.up; i++)
19    for(ORInt j = M.low; j <= M.up; j++)
20       [disjunctive[[resource at: i : j]] add: [task at: i : j]];
21 for(ORInt i=M.low; i <= M.up; i++)
22    [m add: disjunctive[i]];
```

Figure 6.2: High-level technology-independent model in Objective-CP.

duration of each activity on its disjunctive resource and finally adds the disjunctive resources to the model.

Recall that this model is purely descriptive, technology agnostic and captures a triplet $\langle X, C, O\rangle$ in which X is the set of variables, C is the set of constraints and O is an optional objective function. To exploit this model, it is necessary to *concretize* the model into a specific *program*.

Each technology imposes restrictions on what vocabulary can be used to describe models. For instance, a MIP requires linear inequalities over discrete and continuous variables only. Objective-CP uses *model transformations* such as τ to rewrite models into refined forms that are equivalent but conform to the requirements of the technology. Namely, $M_1 = \tau(M_0)$ captures the rewriting of M_0 into an equivalent M_1. Once rewritten, models must be mapped into a solver. Objective-CP achieves this through a *concretization* function γ that delivers an executable optimization program for a technology T, i.e., $P = \gamma_T(\tau(m))$. Refer to chapter 5 for the full details and the formalization of this process. The same high-level model can be concretized several times into multiple solver instances. In particular, Objective-CP supports the simultaneous concretization of one model into both a CP solver and a MIP solver, yielding two *independent* programs.

Scheduling Reformulations The Objective-CP *model reformulations* must be adapted to scheduling. The input is the model presented in Figure 6.2. Three reformulation operators are

$$\texttt{min}\ makespan$$
$$\texttt{s.t.} \begin{cases} precedes(task_{i,j}, task_{i+1,j}) & \forall i \in M, \forall j \in J \\ finished\_by(task_{m,j}, makespan) & \forall j \in J \\ disjunctive(\{task_{\sigma_k^j,j} \mid j \in J, k \in 1 \dots m, \sigma_k^j = r\}) & \forall r \in M \end{cases}$$

Figure 6.3: Global constraint formulation

$$\texttt{min}\ makespan$$
$$\texttt{s.t.} \begin{cases} x_{i,j} \geq 0 & \forall j \in J, \forall i \in M \\ x_{\sigma_h^j,j} \geq x_{\sigma_{h-1}^j,j} + p_{\sigma_{h-1}^j,j} & \forall j \in J, h \in 2, \dots, m \\ x_{i,j} \geq x_{i,k} + p_{i,k} - z_{i,j,k} * V & \forall j, k \in J, k < j, i \in M \\ x_{i,k} \geq x_{i,j} + p_{i,j} - (1 - z_{i,j,k}) * V & \forall j, k \in J, k < j, i \in M \\ makespan \geq x_{\sigma_m^j,j} + p_{\sigma_m^j,j} & \forall j \in J \\ z_{i,j,k} \in \{0, 1\} & \forall i \in M, \forall j \in J, \forall k \in J \end{cases}$$

Figure 6.4: Disjunctive formulation

provided:

- τ_{CP}: Transforms the high-level model into a form suitable for a constraint programming solver supporting global constraints. The resulting model shown in Figure 6.3 maps perfectly to the high-level model.

- $\tau_{MIP-Disjunctive}$: Employs a big-M modelization technique to encode the disjunctive constraints. The application of the operator produces the model shown in Figure 6.4.

- τ_{MIP-TI}: Uses the time-indexed formulation shown in Figure 6.5.

The implementation of the reformulation operators uses rewriting rules for the global constraints similar to those found in [29]. For instance, an operator creates auxiliary variables and visits the global constraints to replace them with linear encoded equivalents (see chapter 9 for implementation details). Below is an example of a rule applied to disjunctive resources to produce a big-M linear formulation:

$$
\begin{array}{l}
\texttt{min } makespan \\
\texttt{s.t.} \begin{cases}
\sum_{t \in H} x_{i,j,t} = 0 & \forall j \in J, \forall i \in M \\
\sum_{t \in H} (t + p_{i,j}) * x_{i,j,t} \leq makespan & \forall j \in J, \forall i \in M \\
\sum_{j \in J} \sum_{t' \in T_{i,j,t}} x_i, j, t' \leq 1 & \forall i \in M, \forall t \in H, \\
 & T_{i,j,t} = \{t - p_{i,j} + 1, \ldots, t\} \\
\sum_{t \in H} (t + p_{\sigma^j_{h-1},j}) * x_{\sigma^j_{h-1},j,t} \leq \sum_{t \in H} t * x_{\sigma^j_h,j,t} & \forall j \in J, h \in 2, \ldots, m \\
x_{i,j,t} \in \{0,1\} & \forall i \in M, \forall j \in J, \forall t \in H
\end{cases}
\end{array}
$$

Figure 6.5: Time-indexed formulation

```
1 linearize(disjunctive) ⇒
2    with: intvar z_{i,j} ∈ {0,1}  ∀ t_i, t_j ∈ tasks(disjunctive), t_i ≠ t_j
3    in:   forall t_i, t_j ∈ tasks(disjunctive) ∧ t_i ≠ t_j :
4             post: start(t_i) + duration(t_i) ≤ start(t_j) + z_{i,j} * M
5             post: start(t_j) + duration(t_j) ≤ start(t_i) + (1 − z_{i,j}) * M
```

Similar rules exist for other *global constraints* such as *precedes* and *finish_by* as well as different rules for the time-indexed formulation.

Custom Search It is worth reiterating that models produced by the reformulation operators above are still purely descriptive and must be concretized into a solver and coupled with a search procedure (when necessary) to obtain *runnables*. The empirical results presented next use two custom search procedures. The search shown in Figure 6.6 depicts a custom procedure as well as how to create a Constraint Programming runnable for model m (from Figure 6.2) with that specific procedure. The code uses the slack that exist on the disjunctive resources to select a machine and sequence the activities of that machine first. Note how the sequencing in lines 9–11 uses a lexicographic heuristic based on earliest start time and earliest completion time to rank the activities.

The second search is the very effective (but incomplete) Large Neighborhood Search [68]. A jobshop-friendly version is taken from [64]. Namely, it is an iterative process in which each iteration limits the number of failures to $3 * |J| * |M|$. When the limit is reached, LNS randomly selects two machines as well as a time window and fixes the precedence that exist between activities *outside* the time window in the incumbent (best) solution. It then re-optimizes.

```
1 ORInt heuristic(id<CPProgram> cp,ORInt i) {
2   return [cp globalSlack: disjunctive[i]] + 1000*[cp localSlack:disjunctive[i]];
3 }
4 ...
5 id<ORRunnable> r0 = [ORFactory CPRunnable:m solve:^(id<CPProgram> program) {
6  [cp forall: M orderedBy:^ORInt(ORInt i) {return heuristic(cp,i);} do:^(ORInt i){
7      id<ORTaskVarArray> t = disjunctive[i].taskVars;
8      [cp sequence: disjunctive[i].successors
9                 by: ^ORDouble(ORInt i) { return [cp est: t[i]]; }
10              then: ^ORDouble(ORInt i) { return [cp ect: t[i]];}];
11  }];
12  [cp label: makespan];
13 }];
```

Figure 6.6: Basic Disjunctive scheduling search procedure.

```
1 id<ORModel> m = ... // Def. of Jobshop Model
2 id<ORModel> LinearModel = [ORFactory linearize:m encoding:Disjunctive];
3 id<ORRunnable> r0 = [ORFactory CPRunnable: m solve: search ];
4 id<ORRunnable> r1 = [ORFactory MIPRunnable: LinearModel];
5 id<ORRunnable> parallel = [ORCombinator completeParallel: r0 with: r1];
6 [parallel run];
```

Figure 6.7: Running a CP and MIP encoding of jobshop in parallel.

6.1.2 Combinators

Figure 6.7 illustrates the handful of lines of code required to create a composite parallel solver. Line 2 creates the chosen linear reformulation. Lines 3 and 4 create the CP and MIP runnable from the original formulation m and the selected linear reformulation $linearModel$[1]. Finally line 5 creates the parallel composite and lines 6 executes the resulting hybrid. Note how all the integration and communication aspects are fully automated. Indeed, the parallel combinator automatically takes care of the necessary plumbing to concurrently share the various products and transcode solutions as needed. Readers are referred to chapter 5 for full details.

6.2 Case Studies

Basic Results Experimental results are provided for various standard instances of the jobshop problem. Some instances remain very difficult to solve to optimality, even for modest sizes. Results are presented on various solvers described below:

MIP Gurobi 6.04 MIP with 2 threads and a disjunctive encoding.

[1]Line 3 refers to the search procedure defined earlier with a closure and named *search*.

CP Objective-CP solver with 2 threads and a common *global slack* heuristic.

$CP \parallel MIP$ A parallel composite with CP and MIP solvers.

$LNS_{CP} \parallel MIP$ A parallel composite with a CP-based LNS and a MIP solver.

$LNS_{CP} \parallel CP$ A parallel composite with a CP-based LNS and a plain CP solver.

The time-indexed formulation is omitted as, consistent with [50], it is not competitive and only runs on small instances. All experiments are run on Mac OS X 10.10.5 with 4 GB or memory and an Intel Core 2 Duo 2.13 GHz. Table 6.1 shows the best upper bound achieved and the running time in seconds for each solver within 10 minutes. The MIP solver uses, by default, 2 threads. For fairness, the CP solver uses a parallel tree search with 2 threads too. Composite solvers use 1 thread for the first runnable and 2 threads for the second. For instance, $CP \parallel MIP$ uses 1 thread for CP and 2 for the MIP.

Figure 6.8 augments Figure 6.1 found earlier in this chapter with a new parallel solver combining CP and MIP. The results are interesting, suggesting that combining the MIP and CP solvers yields a much more robust solver which is capable of performing about as well as the better of the two independent solvers. On several instances, the hybrid performs better than either of the standalone solvers (`Orb7, Orb8, Orb10`).

A full table of results, Table 6.1, confirms that MIP and CP are competitive as reported in [50] with MIP outperforming CP quality-wise (`la31,la36,la37,la38,la21`) and timewise (`Orb05,Orb08`) on 7 of the 15 benchmarks. More interestingly, the composite, $CP \parallel MIP$, improves the quality of the solution over the standalone CP on all the `la` instances. The $LNS_{CP} \parallel MIP$ composite does even better managing to close `la31,la37` and delivering the best incumbent on `la38`. Finally, the $LNS_{CP} \parallel CP$ composite is the best hybrid of this pack. It yields high quality bounds from the LNS search and restores completeness through its reliance on a complete parallel CP search. The running times are often the best and this composite now closes `la36` and further improves `la40`.

Figure 6.9 visualizes the running time of all the solvers on the `Orb` instances (same data as Table 6.1).

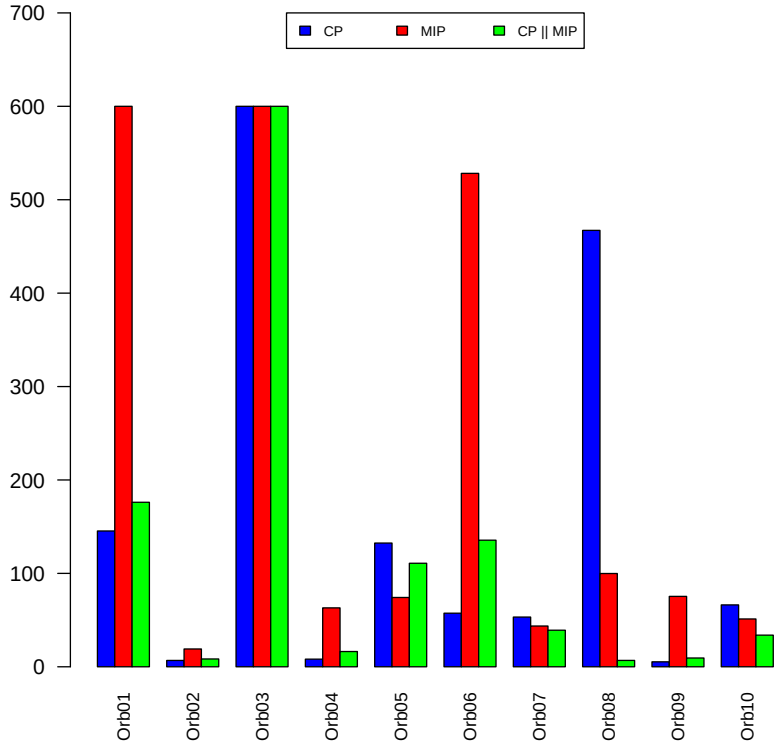

Figure 6.8: Running time in seconds of independent CP & MIP solvers as well as a parallel CP/MIP hybrid.

Hybrid Communication Table 6.2 reports the number of bounds and solutions exchanged between parallel solvers within the three hybrid runnables on the `Orb` instances. The data shows substantial inter-solver communication among all three hybrids. The MIP solver generally receives more bounds than it generates from both CP and LNS_{CP}. The CP-LNS_{CP} hybrid shows a mixed story with roughly equal bound generation on some instances and one dominant solver (not always the same) on others.

Figures 6.10, 6.11 and 6.12 provide a more detailed look at the data from Table 6.2, providing time points as bounds are received between parallel solvers during the search. For many of the instances in these figures, bounds are exchanged in a flurry of activity over several seconds (with both solvers actively contributing) before the hybrid enters a prolonged period of stagnation. Consider `Orb01` in Figure 6.10, bounds are exchanged for only about 8s, while Table 6.1 reports this solver

Instances	CP		MIP		$CP \parallel MIP$		$LNS_{CP} \parallel MIP$		$LNS_{CP} \parallel CP$	
	time	ub	time	ub	time	ub	time	ub	time	ub
Orb01(10×10)	145.38	1059*	600.0	1072	176.12	1059*	600.0	1071	**41.96**	**1059***
Orb02(10×10)	6.80	888*	19.06	888*	8.36	888*	18.97	888*	**6.33**	**888***
Orb03(10×10)	**600.0**	**1015**	600.0	1021	**600.0**	**1015**	**600.0**	**1005**	600.0	1015
Orb04(10×10)	8.17	1005*	63.07	1005*	16.33	1005*	53.33	1005*	**7.67**	**1005***
Orb05(10×10)	132.46	887*	74.20	887*	110.82	887*	70.92	887*	**70.35**	**887***
Orb06(10×10)	**57.37**	**1010***	528.22	1010*	135.53	1010*	600.0	1010**	52.05	1010
Orb07(10×10)	53.22	397*	43.64	397*	39.15	397*	18.65	397*	**11.23**	**397***
Orb08(10×10)	467.19	899*	99.86	899*	6.82	899*	84.41	899*	**4.57**	**899***
Orb09(10×10)	**5.31**	**934***	75.36	934*	9.41	934*	85.55	934*	**5.31**	**934***
Orb10(10×10)	66.24	944*	51.20	944*	33.87	944*	28.34	944*	**5.31**	**944***
la31(30×10)	600.0	2801	600.0	2003	600.0	2109	30.82	1784*	**17.23**	**1784***
la36(15×15)	600.0	2059	600.0	1292	600.0	1297	600.0	1281	**136.96**	**1268***
la37(15×15)	600.0	1855	600.0	1454	600.0	1478	**13.62**	**1397***	13.97	1397*
la38(15×15)	600.0	1633	600.0	1230	600.0	1243	**600.0**	**1196**	600.0	1255
la21(15×10)	600.0	1129	600.0	1079	600.0	1097	600.0	1058	**600.0**	**1046**

Table 6.1: Experimental Results for CP and MIP solvers as well as three hybrids. A star (*) indicates the optimal bound was found and proved.

	$LNS_{CP} \parallel MIP$		$CP \parallel MIP$		$LNS_{CP} \parallel CP$	
	MIP from LNS	LNS from MIP	MIP from CP	CP from MIP	CP from LNS	LNS from CP
orb01	180	20	81	49	77	24
orb02	172	28	36	9	6	39
orb03	224	52	57	20	35	41
orb04	96	26	131	14	52	79
orb05	73	40	89	28	49	56
orb06	152	38	55	16	16	48
orb07	62	16	63	21	43	33
orb08	223	40	197	50	66	107
orb09	86	32	78	30	37	66
orb10	144	28	86	36	58	47

Table 6.2: The number of bounds / solutions exchanged between parallel solvers.

(different run) finished around 42s. The discrepancy represents the time the *CP* solver spent proving optimality once an optimum had been found. Compare this with the hybrid $LNS_{CP} \parallel MIP$ in Figure 6.11 on the same instance. In this case, bounds are only shared for about 3.5 seconds before the solver stagnates and ends up timing out at 600s without having found an optimal solution.

There are also a number of instances in Figure 6.12 (`Orb01, Orb03, Orb05, Orb07`) which show different behavior. Rather than a a single, quick burst of activity early in the search, we can see several waves of activity emerging over a period of a minute or two. Overall the hybrid communication data demonstrates a variety of interaction profiles which vary not only between

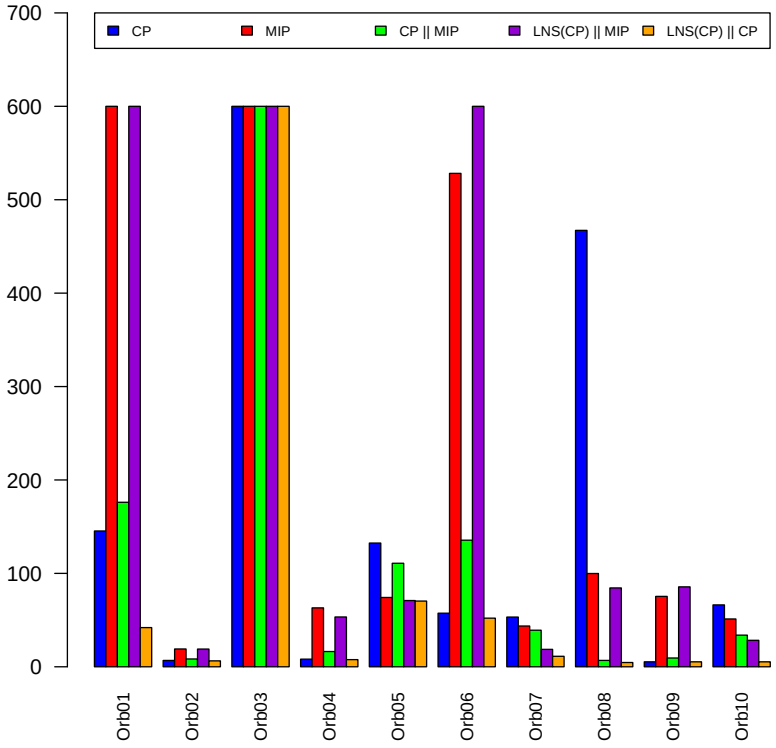

Figure 6.9: Comparison of standalone CP & MIP solvers with three cooperative hybrid solvers.

solvers, but also between instances.

Robust Runnables What is, perhaps, unexpected in Table 6.1 is the behavior on benchmark like `Orb08` where CP takes 467 seconds to prove optimality, MIP requires 99 seconds for the same result while the $CP \parallel MIP$ composite completes in a mere 7 seconds. The explanation lies in the parallel search. Conventional wisdom dictates that the number of threads ought to be equal to the number of cores. When CP (or MIP) is executing alone, it carries out a parallel tree search with 2 threads. When executing in the composite, the CP solver uses a sequential search while the MIP uses a parallel search (with 2 threads). The observed behavior is a simple lack of robustness of the parallel tree search. When the optimum is found, CP can prove optimality near instantly. Finding the optimum however, proves difficult. If a node on the path from the root to that optimum is

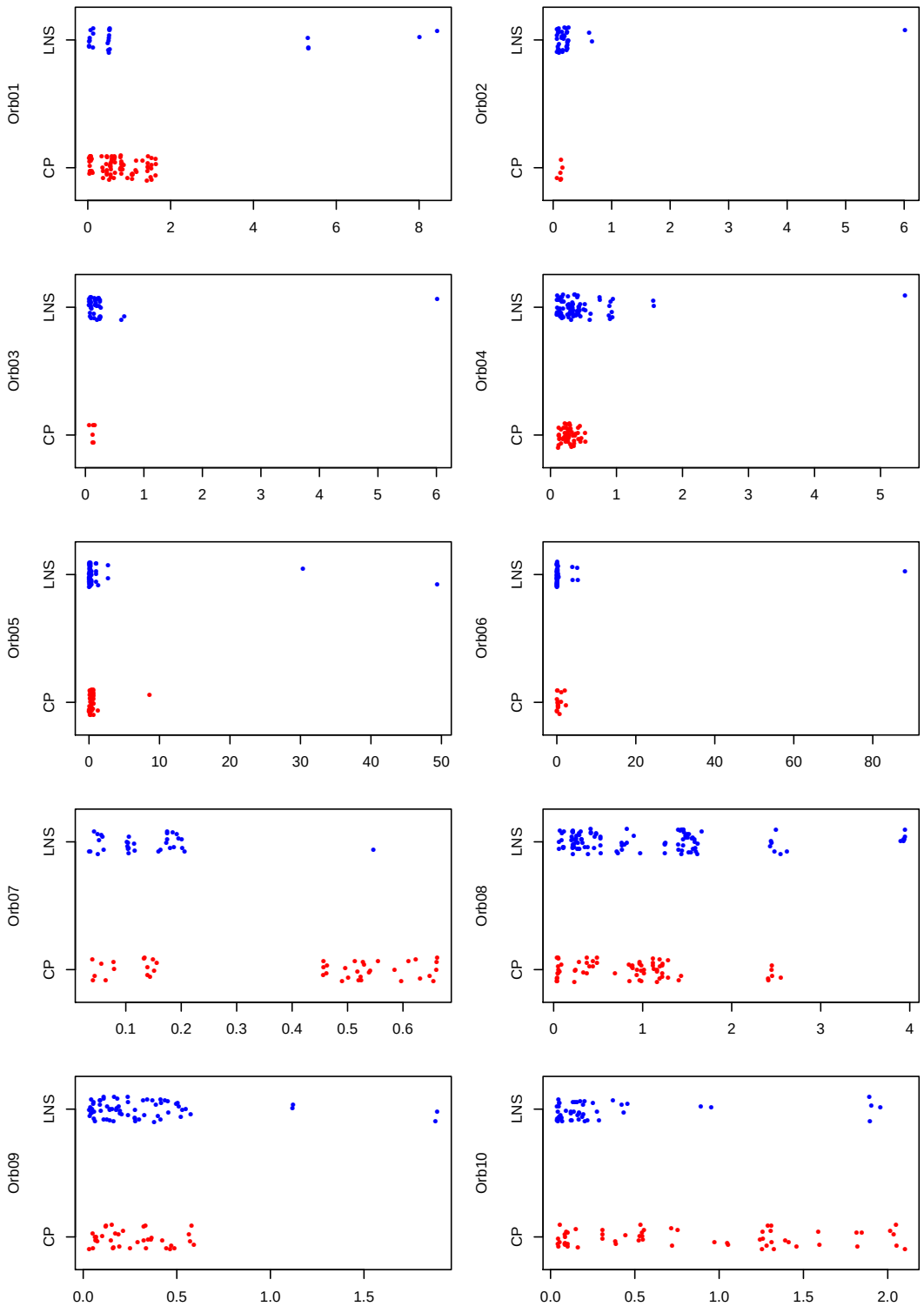

Figure 6.10: Time points (in seconds) as CP and LNS_{CP} receive bounds

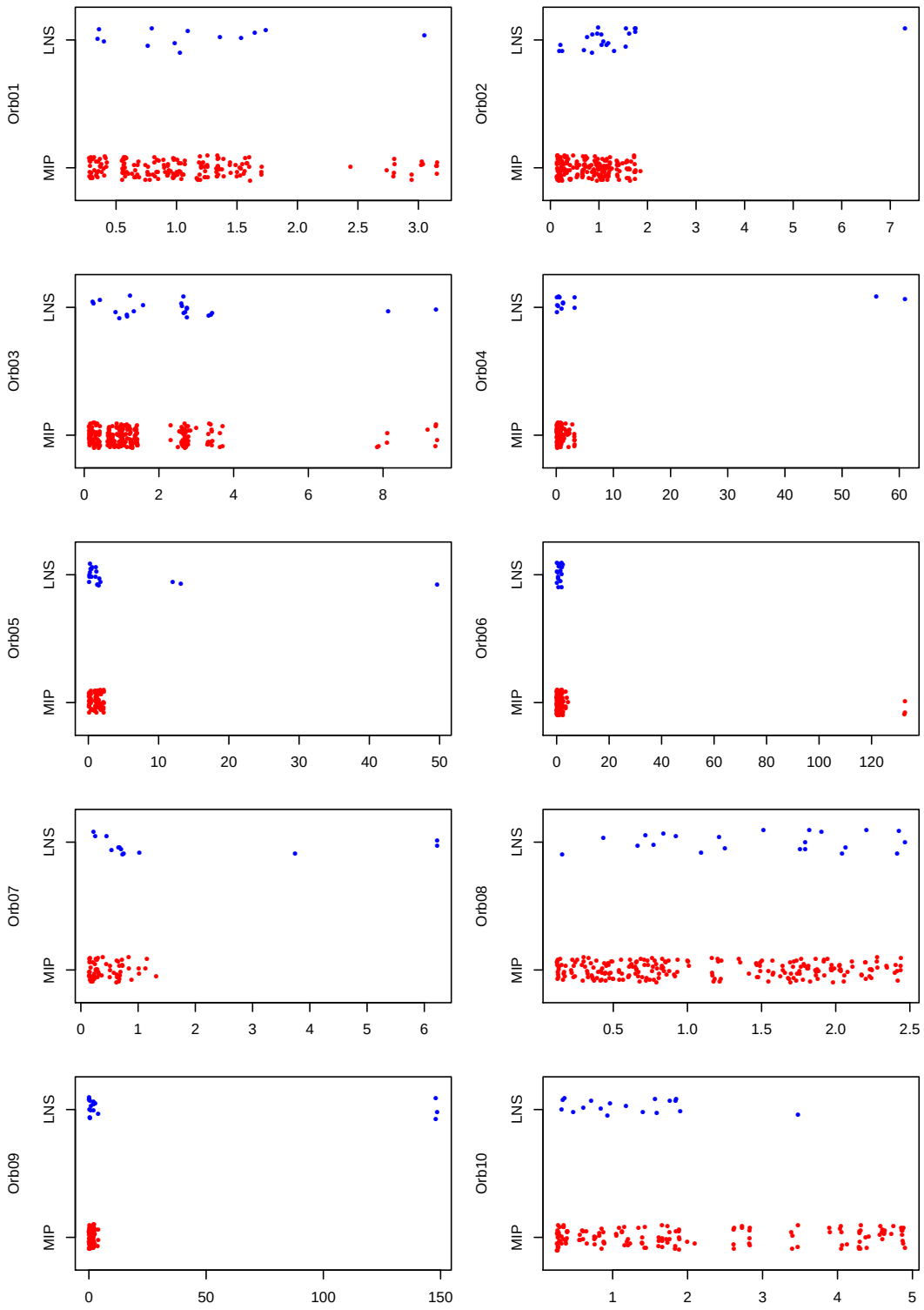

Figure 6.11: Time points (in seconds) as MIP and LNS_{CP} receive bounds

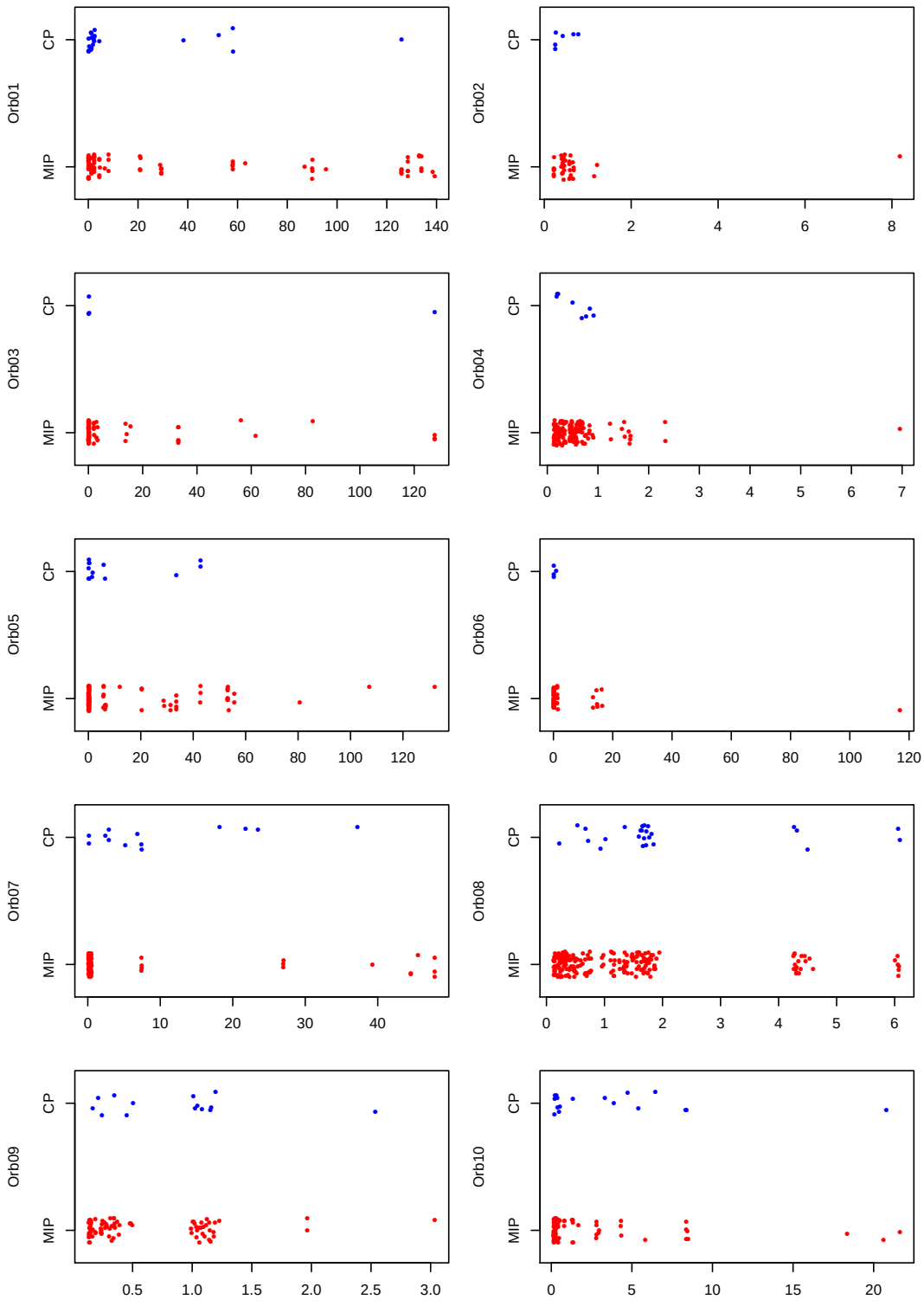

Figure 6.12: Time points (in seconds) as CP and MIP receive bounds

Instances	CP			MIP			CPS
threads	1	2	4	1	2	4	3
Orb05 (10 × 10)	70.20	75.26	29.58	32.92	43.43	17.82	60.36
Orb07 (10 × 10)	38.60	48.65	9.7	40.64	26.46	34.56	39.12
Orb08 (10 × 10)	1.66	600	600	123.28	55.64	68.70	1.28
Orb10 (10 × 10)	28.06	32.07	29.52	65.80	30.47	15.49	24.84
la10 (15 × 5)	0.18	0.22	0.16	600	600	600	0.26
la11 (20 × 5)	1.46	2.38	2.21	600	600	600	1.22

Table 6.3: Performance of CP and MIP jobshop solvers given 1, 2, and 4 threads.

shared with other threads, the discovery of the optimum may be postponed until that node is stolen, and a substantial delay may be incurred.

This phenomenon happens within MIP solvers too and is illustrated in Table 6.3 where the data was collected on a quad-core MacPro with a Xeon at 3.2Ghz running OSX 10.11. For instance, the solving time for MIP on `Orb10` improves as threads are added while it barely moves for the CP solver while `Orb05` and `Orb07` experience the opposite effect (adding threads hurt Gurobi). To explore this fairness question Table 6.3 reports on a few instances involving the CP and MIP solvers with 1, 2 and 4 threads as well as a new composite, dubbed CPS, which composes a sequential CP solver with a parallel tree search CP solver. The number of threads can have unsettling effects, sometimes improving or worsening the solving time. The ability to use the composite $CP \parallel CP(2)$ alleviates the problem. Indeed, sequential and parallel CP share their bounds.

Chapter 7

Application: Large Neighborhood Search

This purpose of this chapter is twofold, first a new Large-Neighborhood Search heuristic which proves very useful for IMRT (section 1.2.5) is introduced and second, a demonstration of how this search can be effectively automated (using *CML* in this case). The IMRT problem is solved in this chapter using the *Counter Model* and a CP solver. The *Counter Model* no longer represents the state of the art on some instance classes (problem instances with relatively small maximum *beam on* times), as a new CP-LP hybrid model based on the *ShortestPath* constraint has demonstrated superior results [17]. The *ShortestPath* hybrid approach, however, is currently limited by the fact that the number of variables increases exponentially with the maximum *beam on* time. As new generations of multileaf collimator are developed, it is likely that larger instances will become common.

The aim of this chapter is to provide heuristic search techniques which are orthogonal to the modeling technologies being used. The *Counter Model* is used as the basis for these heuristics as it is quite simple to implement in standard CP environments (in this case COMET) and still shows relatively good performance in comparison to more traditional techniques. In contrast, the *ShortestPath* hybrid approach relies on a non-standard constraint and a more sophisticated implementation requiring bounds passing between LP and CP models. The search heuristics presented

here allow the simpler *Counter Model* to provide quality solutions on instances comparable to those handled by the *ShortestPath* hybrid.

The *Counter Model* has previously been solved using a complete search to find an optimal value for K and prove optimality (refer to problem description in section 1.2.5). This method of exhaustive search quickly becomes impractical as the problem size increases and it often fails to return any solutions within a reasonable time frame. For these larger instances it is often possible to find very good solutions quickly using local neighborhood search heuristics, in particular, that of the *Large Neighborhood Search* (LNS). LNS refers to a family of techniques that considers a large neighborhood definition.

In the case of the *Counter Model* we may define a neighborhood as a subtree of the larger search tree achieved by fixing some subset of the N variables. By fixing some of the Ns, the size of the search tree can be greatly reduced. It then becomes possible to quickly perform a complete search on the reduced tree to find feasible solutions. When a feasible solution is found, a constraint is added bounding the value of K to the best known value.

The effectiveness of the LNS algorithm is determined by the number of N variables that are fixed and the appropriateness of the values to which they are fixed. In the case of the *Counter Model*, it is possible find a heuristic that works very well on a large number of instances. A given $N[i]$ represents the number of C1 matrices in the final decomposition with associated *beam on* time i. It is typical that a C1 decomposition is dominated by C1 matrices with relatively low associated *beam on* times. In practice N variables associated with the highest *beam on* times are often 0 or very close to it. A surprisingly effective heuristic involves *profiling* the values taken by the N's across a large number of instance solutions and using this profiling data as a training set for selecting effective LNS neighborhoods. This training data allows N variables to be fixed to values chosen from a range that is likely to give quality solutions. The LNS search is quite simple and is described in Figure 7.1.

The *timeout* parameter in the above search is user specified as it is impossible to know when an optimal solution has been found with a local search approach. The *split* function partitions the N variables into two subsets N_f and N_s, specifying variables to be fixed (N_f) and variables left unbound for the *Counter Model* search (N_s). Both the *split* and *fix* function will be described in

repeat
$< N_f, N_s >=$ **split**(N)

fix(N_f)
counterSearch(N_s)
until *timeout*

Figure 7.1: Large Neighborhood Search heuristic

function *counterSearch*(N_s):
search by branch-and-bound:

instantiate N_s by lower half first bisection
$S := [1..n]$
while $S \neq \emptyset$
choose row i in S maximizing $\sum_{j=1}^{n} inc(I_{i,j-1}, I_{i,j})$ (row hardness)
$S := S - \{i\}$
for $j := 1$ to m:
for $b := 1$ to $\bar{b}$:
instantiate $Q_{b,i,j}$ by lower half first bisection
on failure break and return to last choice of N_b

Figure 7.2: Search procedure used in the *Counter Model*

future sections. The *counterSearch* function is presented in [10] and is described in

7.1 Profiling

In order to fix the N variables to effective values in the LNS search, it is useful to profile their mean and standard deviation for solutions to randomly generated instances. It has become common to benchmark the Radiation problem against purely random intensity matrices with entries drawn from some uniform distribution. Although such uniform random matrices may be sufficient for benchmarking purposes, it is not obvious that solutions to such problems are similar to real world instances. Real world instances are typically generated from a set of target and critical structures [7]. Targeted structures correspond to diseased tissue while critical structures correspond to vital organs which must receive little or no radiation. It is reasonable to expect a real world intensity map will exhibit some structure related to the contiguous nature of the target and critical structures from which it was derived. Hence, it is essential to differentiate between different classes of instances

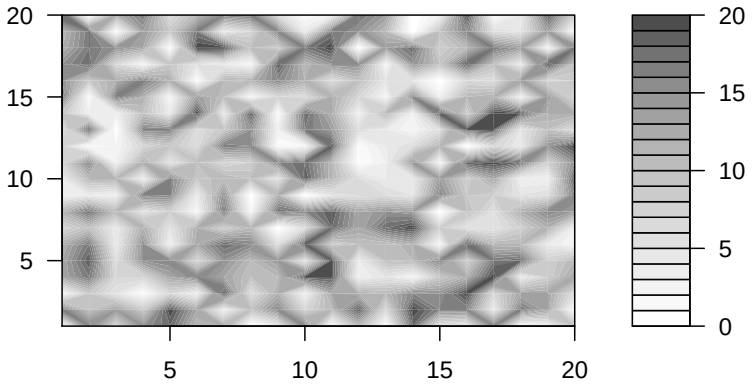

Figure 7.3: Level plot of a typical $20 \times 20 \times 20$ uniform random instance

when doing solution profiling.

Two classes of randomly generated instances will be considered here, the first is the uniformly random intensity matrix and the second is a structured random matrix which exhibits some of the contiguous structure that is expected in real world data. Although there are many similarities between the profiling done on the two classes of random instances, it will be shown that there are significant differences as well. Hence, it is important that profiling is done on a similar class of instances to those being solved for the best results.

7.1.1 Uniform Random Instances

A uniform random instance of size $n \times m \times k$ is an $n \times m$ matrix with entries chosen at random, uniformly from the set $0..k$. Figure 7.3 gives an example of a typical uniform random instance.

Figure 7.4 shows several box plots illustrating how the percentage of the total *beam on* time is distributed among the N variables for several instance sizes. Box plots display the median value (horizontal solid line), the upper and lower quartile (bounded by shaded box), the maximum and minimum observations (bounded by dotted line) and outliers (points). The plots are generated from 50 runs and the percentages are computed using the formula $\frac{i \times N[i]}{B^*}, i \in 1..k$

The plots illustrate that solutions have a clear profile with a relatively small deviation. It also illustrates that matrix size doesn't play a significant role in determining what the solution looks

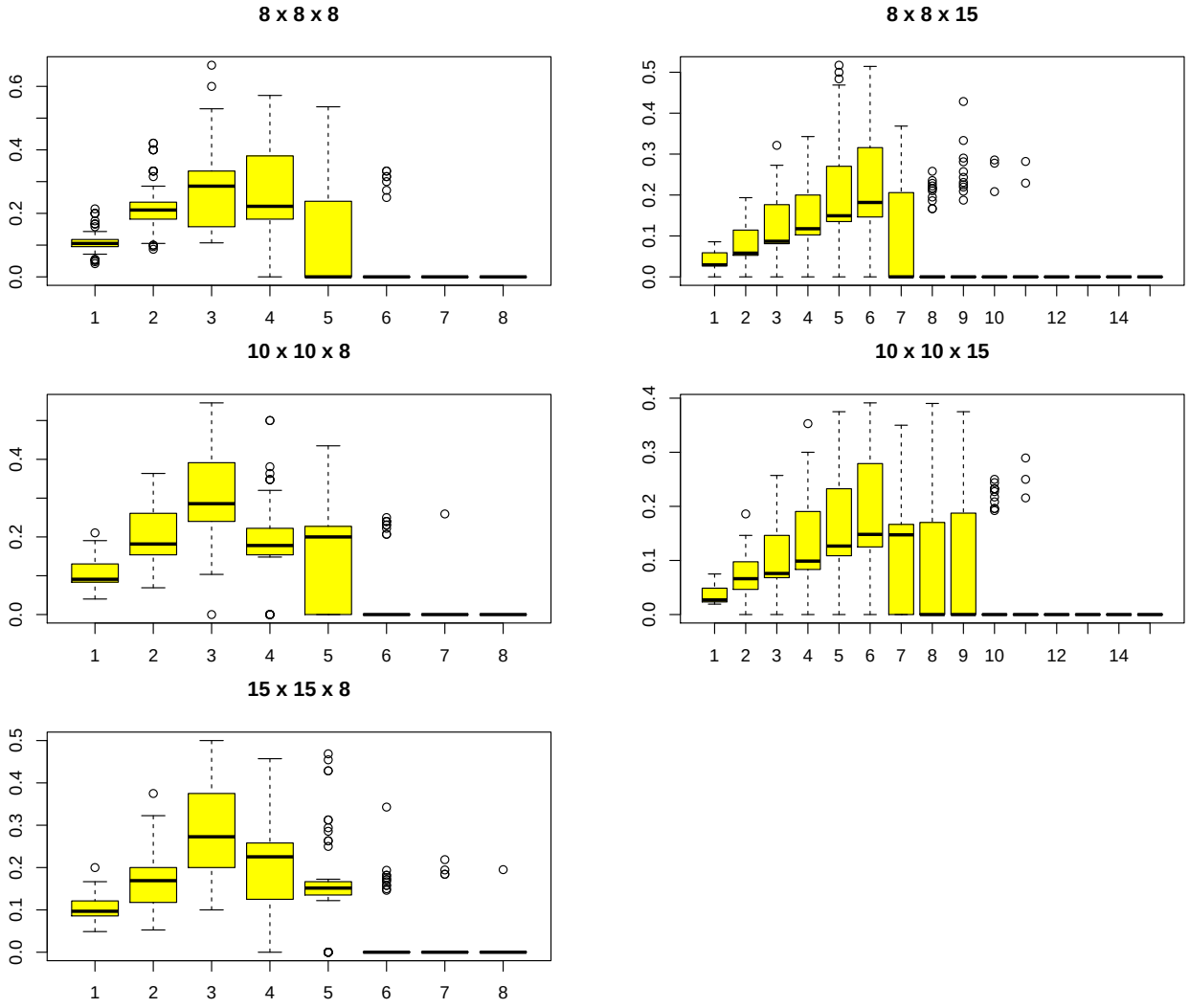

Figure 7.4: values of N variables for solutions to uniformly random instances. Profiling instances of size $15 \times 15 \times 15$ did not finish within a 24 hour time limit.

like. The size of the maximum entry in the matrix, however, does skew the solution profile. This suggests that given a domain for our intensity matrix, we may profile solutions on relatively small matrices and then apply the profile to a LNS search on a larger matrix.

7.1.2 Structured Random Instances

Structured random instances are a different class of randomly generated intensity map that are designed to exhibit some of the local structure found in real world intensity map data. Compare Figure 7.5 to a real world clinical instance in Figure 7.6. The purpose of the uniform random

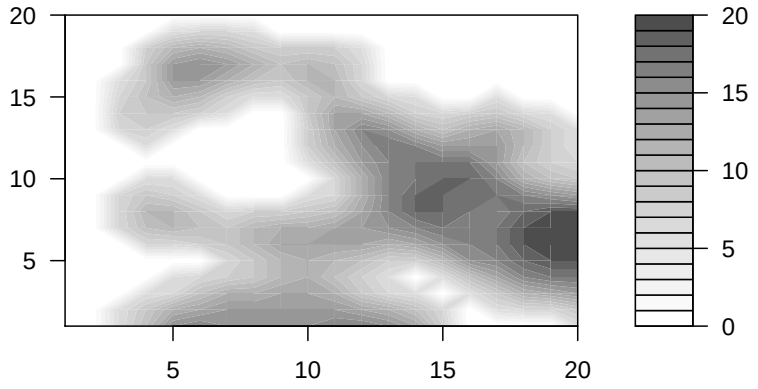

Figure 7.5: Level plots of a typical $20 \times 20 \times 20$ structured random instance

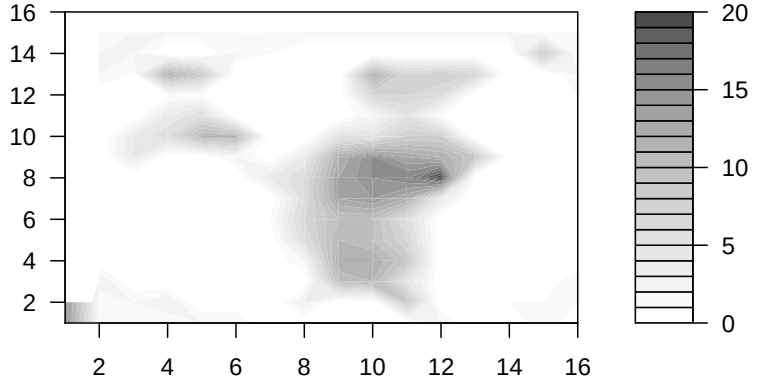

Figure 7.6: Level plots of a typical $15 \times 15 \times 20$ clinical instance

instances is not so much to argue that these structured instances are a substitute for real world instances, but instead to illustrate that different classes of instances can have significantly different profiling characteristics that need to be considered when developing a search strategy. The structured random matrices are generated with the following steps:

1. Begin with a square zero matrix of size $n \times n$

2. Repeat $n/2 - 1$ times: select beam intensity $b \in \frac{B^*}{2}..B^*$. Perform a random walk of length $\frac{n^2}{4}$ labeling entries with intensity b along the walk.

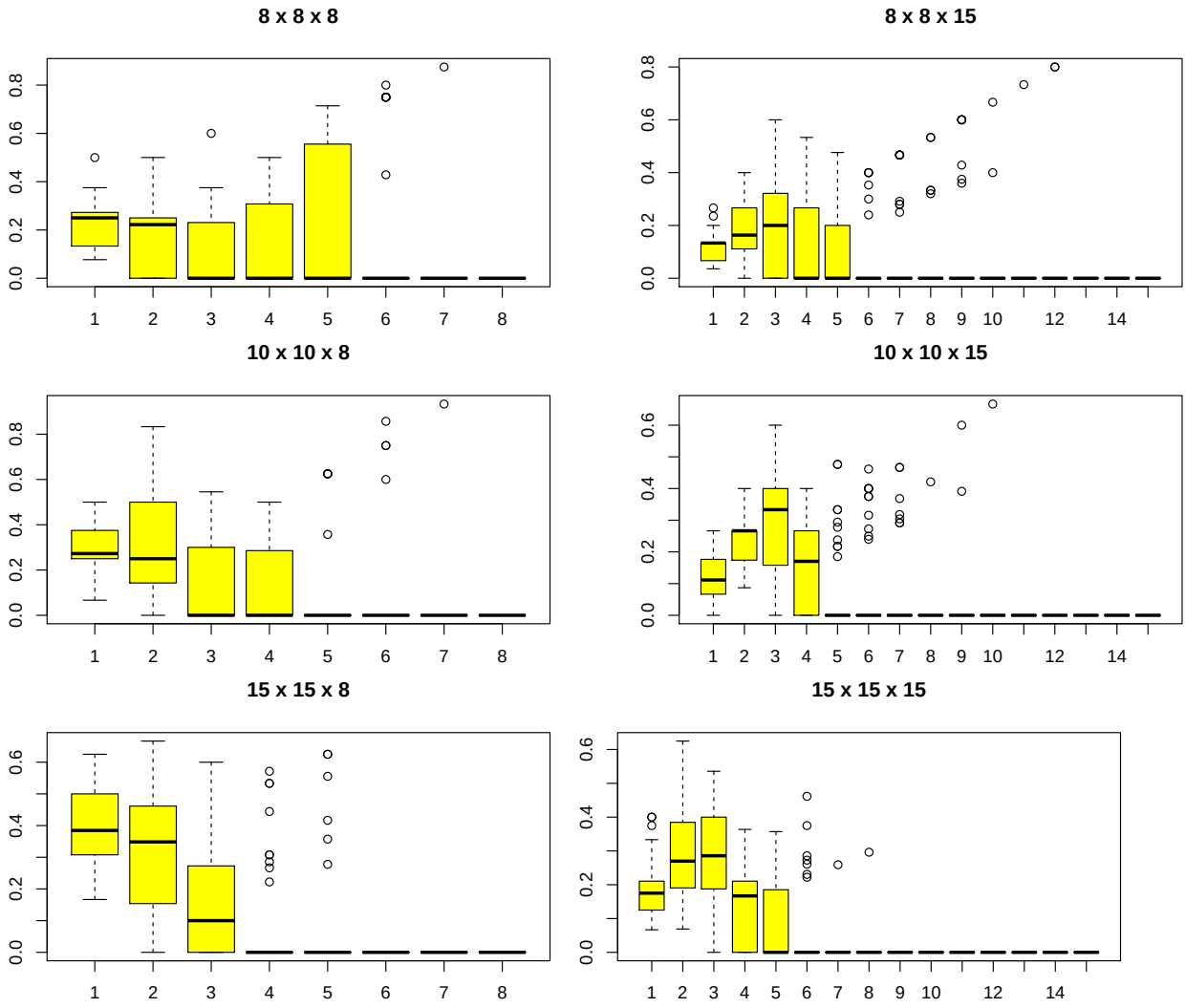

Figure 7.7: values of N variables for solutions to structured random instances

3. Perform a final random walk of length $\frac{n^2}{4}$ using intensity B^*

4. For every entry in the matrix, perform a smoothing by averaging the entry value with its four immediate neighbors.

5. If intensity B^* does not appear in the matrix, let δ be the difference between B^* and the largest value in the matrix. Add δ to all non-zero matrix entries.

The box plots demonstrate that the solution profile of the structured matrices are broadly similar to those of the uniformly random matrices, but exhibit important differences. The most substantial difference is that the structured instances are more heavily dependent on shapes with

low intensity *beam on* times. Hence low indices of N (such $N[1]..N[3]$ for $m \times n \times 8$ instances) have both larger median values and larger standard deviation than the uniformly random instances. Both classes of instances however demonstrate that beam energy is heavily skewed towards the first half of the N variables with the second half of the N variables contributing very little of the beam energy. This indicates that a good strategy is to fix the latter N's and leave the first few unfixed.

7.2 Generating Neighborhoods from Profiling Data

Building a search strategy from the profiling data is straightforward:

Example: Suppose we have an intensity matrix of size $n \times m \times 8$ with a total *beam on* time $B^* = 21$ and the following profile data,

$$\text{mean}[0.184, 0.175, 0.156, 0.148, 0.119, 0.085, 0.052, 0.025],$$
$$\text{stddev}[0.028, 0.078, 0.117, 0.145, 0.114, 0.106, 0.061, 0.042]$$

The values in the *mean* array are percentages of the total *beam on* energy B^* to be delivered by the term $i \times N[i]$ (recall $B^* = \sum_{i=1}^{\bar{b}} i \times N[i]$). Hence, the expected value of a given $N[i]$ is given by the equation $N[i] = \frac{B^* \times mean[i]}{i}$

Applying this equation, the expected values of the N variables are:

$$[3.864, 1.838, 1.092, 0.777, 0.500, 0.298, 0.156, 0.066]$$

Using the same formula for $N[i]$ and the standard deviation array, it is possible to create arrays of ranges representing any number of standard deviations from the expected value.

$$\text{1SD}[3..5, 1..2, 0..2, 0..1, 0..1, 0..1, 0..1, 0..1] \text{ (1 standard deviation)}$$
$$\text{2SD}[2..6, 0..4, 0..3, 0..3, 0..2, 0..2, 0..1, 0..1] \text{ (2 standard deviations)}$$

Recall that one standard deviation from the mean will include 68% of values while two standard deviations covers 95%. Hence, the 2SD array gives us safe ranges from which we can confidently choose reasonable values for fixing corresponding N variables, with one exception. The 2SD array is derived from data from 50 runs and, hence, isn't necessarily indicative of what a typical solution

function $split(N)$:
$N_f = \{\}$
$s \leftarrow 0$
for $i = \bar{b} \rightarrow 1$ **do**
    $append(N_f, N[i])$
    $s \leftarrow s + median(N[i])$
    if($s > 0.5$) **break**
end forreturn $< N_f, N - N_f >$

Figure 7.8: The *split* function determines a subset of the N variables to be fixed.

looks like for a single run. On a given single run, it is quite rare for more than a single variable in the range $N[\frac{\bar{b}}{2}]..N[\bar{b}]$ to be non-zero. In fact, it is often the case that all of these variables are 0. Fixing variables uniformly using 2SD does not reflect this fact. A better strategy is to permit at most one of the variables $N[\frac{\bar{b}}{2}]..N[\bar{b}]$ to take on non-zero values with a relatively small probability.

We are now in a position to discuss implementations of the *split* and *fix* functions. Note that both *split* and *fix* implement heuristic approaches that worked well in practice on the instances tested, but it often possible to achieve better results on specific instances by modifying the parameters. For example, the cardinality of the fixed set can be increased for faster performance, but this generally results in lower quality solutions. Hence, on medium sized intensity maps (say $16 \times 16 \times 10$) it is often advantageous to have a smaller set of fixed variables but on very large instances it will likely be necessary to fix nearly all the variables in order to find any solutions within a reasonable time frame. The heuristic used in the *split* function shown in Figure 7.8 makes use of the profiling data. The idea is that the N variables with higher indices generally have a much higher standard deviation than the N's with lower indices and, therefore, it makes sense to start fixing these variables first. The heuristic fixes N's starting with the highest indices and works its way down until no more than 50% of the total beam intensity has been fixed (based on median values). The value 50% is a parameter that may be modified for better results on particular instances or other classes of instances.

Notice that based on the profiling data presented in figure 7.4, instances of size $n \times n \times 8$ will fix all N variables besides $N[1]$ and $N[2]$. Figure 7.9 shows the *fix* function is presented. This function assumes that a set of uniform distributions $\{D_n\}_{n \in N}$ has been created based on profiling data (such as the 2SD array above).

function $fix(N_f)$:
for $n \in N_f$ **do**
  if $index(n) < \frac{\bar{b}}{2}$ **then** $n \leftarrow random(D_n)$
  else $n \leftarrow 0$
end for

choose i **from** $1..\bar{b}$ **uniformly**
if $i \geq \frac{\bar{b}}{2}$ **and** $N[i] \in N_f$ **then**
  $N[i] \leftarrow random(D_{N[i]})$
end if

Figure 7.9: The *fix* function fixes variables using profiling data.

7.3 Automating LNS in CML

```
1 import "lib/LNS";
2 // LOAD intensity matrix ...
3 bt\_max = (all(i in rows, j in cols) intensity[i,j]).max();
4 ints_sum = sum(i in rows, j in cols) intensity[i,j];
5 btimes = 1..bt_max;
6 # Pre-compute optimal beam-on time
7 beam_time = 0;
8 forall(i in cols) {
9         v = intensity[i, 1] + sum(j in 2..n) max(intensity[i, j] - intensity[i, j-1], 0);
10        if(v > beam_time) beam_time = v;
11 }
12 model Radiation {
13    var{int} K(0..m*n);
14    var{int} N[btimes](0..m*n);
15    var{int} Q[rows, cols, btimes](0..m*n);
16    objective: Minimize(K);
17    post: beam_time == sum(b in btimes) b * N[b];
18    post: K == sum(b in btimes) N[b];
19
20    forall(i in rows, j in cols)
21      post: intensity[i,j] == sum(b in btimes) b * Q[i,j,b];
22    forall(i in rows, b in btimes)
23      post:  N[b] >= Q[i,1,b] + sum(j in 2..n) max(Q[i,j,b] - Q[i,j-1,b], 0);
24 }
25
26 cpm = StdLNS(CP(Radiation));
27 whenever cpm@freeze() ` forall(i in ${floor(bt_max/2)..bt_max}) label(N[i], 0); `
28 whenever cpm@searchFragment() `
29    forall(b in ${btimes}: !N[b].bound()) {
30       while(!N[b].bound()) {
31          int mid = (N[b].getMin() + N[b].getMax())/2;
32          try<cpm> cpm.lthen(N[b],mid+1); | cpm.gthen(N[b],mid);
33 } } `
34 cpm.emit_comet_file("Radiation.CP.co");
```

Figure 7.10: IMRT counter model in CML with LNS search

The CML model for the Intensity-Modulated Radiation Therapy problem (IMRT) demonstrates how sophisticated LNS searches can be easily obtained. Figure 7.10 shows a counter model formulation of the IMRT problem using a the custom LNS search presented in this chapter. In the listing

below, variables are frozen on line 27. A custom search over the fragment is then provided on lines First the abstract model is concretized using the CP operator, then the IBS operator applies an Impact Based Search to the model and finally the StdLNS operator automatically picks up the Impact Based Search and uses it when searching over the *active fragment*. A custom search over the fragment can also be provided on lines 28-33.

7.4 Case Study

The LNS approach runs substantially faster than a complete *Counter Model* search on larger instances producing either the optimal or near optimal solution in a majority of cases. In the figures below we consider how frequently the LNS search produces the optimal solution and how frequently it produces a solution within 10% of the optimal (near optimal) over various instance sizes (50 runs per size). The time required for a complete search on a given instance is used as the time limit for the LNS search. Additionally, the figures illustrate the average time required to find the solution as a fraction of the average time taken by the complete search. These time plots only consider LNS instances which find solutions within the given time limit.

Figures are provided for both uniform and structured random instances of size $n \times n \times k$ for $k = 8$ and $k = 15$. In the uniform case, when $k = 8$, $n \in 6..18$ and when $k = 15$, $n \in 6..12$. Similarly for structured instances, when $k = 8$, $n \in 6..28$ and when $k = 15$, $n \in 6..18$. Larger ranges are considered for the structured instances as the complete search finishes relatively fast (compared to uniform instances) since large segments of the matrix are zeros.

Figure 7.11 shows that on uniform instances where $k = 8$, LNS only produces the optimal solution about 20% to 30% of the time without much improvement seen on larger instances. When $k = 15$ performance is slightly better with the optimal solution being found about 50% of the time ($n = 12$). The time required for the LNS search when it finds the optimal solution approaches about 20% of the time for the complete search for $k = 8$ and 10% for $k = 15$.

When we consider solutions within 10% of the optimal (near optimal) the results are more impressive. As the instance size grows the LNS is able to find a near optimal solution over 80% of the time when $k = 8$ and 90% when $k = 15$. Furthermore, we see the time required for the LNS

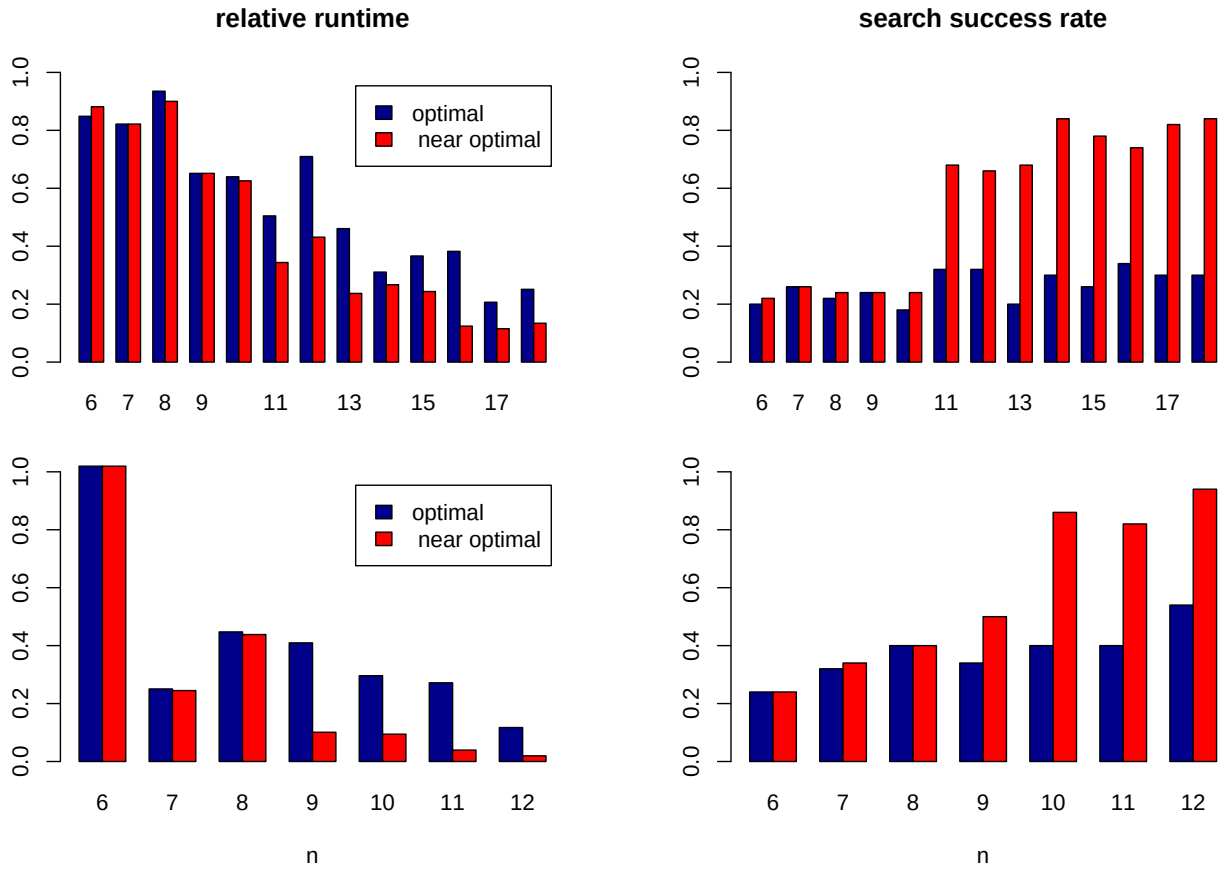

Figure 7.11: Uniform random instances of sizes $n \times n \times 8$ (top row) and $n \times n \times 15$ (bottom row)

search relative to the complete search drops considerably (around 10% for $k = 8$ and $< 5\%$ for $k = 15$) as the instance size grows. The jump in the search success percentage (see jump between $n = 10$ and $n = 11$ when $k = 8$) reflects the fact that for smaller instances, solution values were so small that the search had to produce the optimal solution to be within the 10% margin.

Figure 7.12 shows the LNS was less successful on structured instances. In this case the search produced the optimal solution nearly 40% of the time when $k = 8$ and 30% of the time when $k = 15$. The near optimal solution is produced about 60% for $k = 8$ and 50% of the time when $k = 15$. The relative runtime for both the optimal and near optimal solutions is trending towards 20%, but with several large 'spikes'. Notice that the LNS search is not actually performing worse when $k = 15$ relative to $k = 8$. Instead, the performance difference reflects the fact that much larger instances can be considered for the smaller k providing the LNS search greater opportunity to differentiate from the complete search. As mentioned above, the LNS search performs better

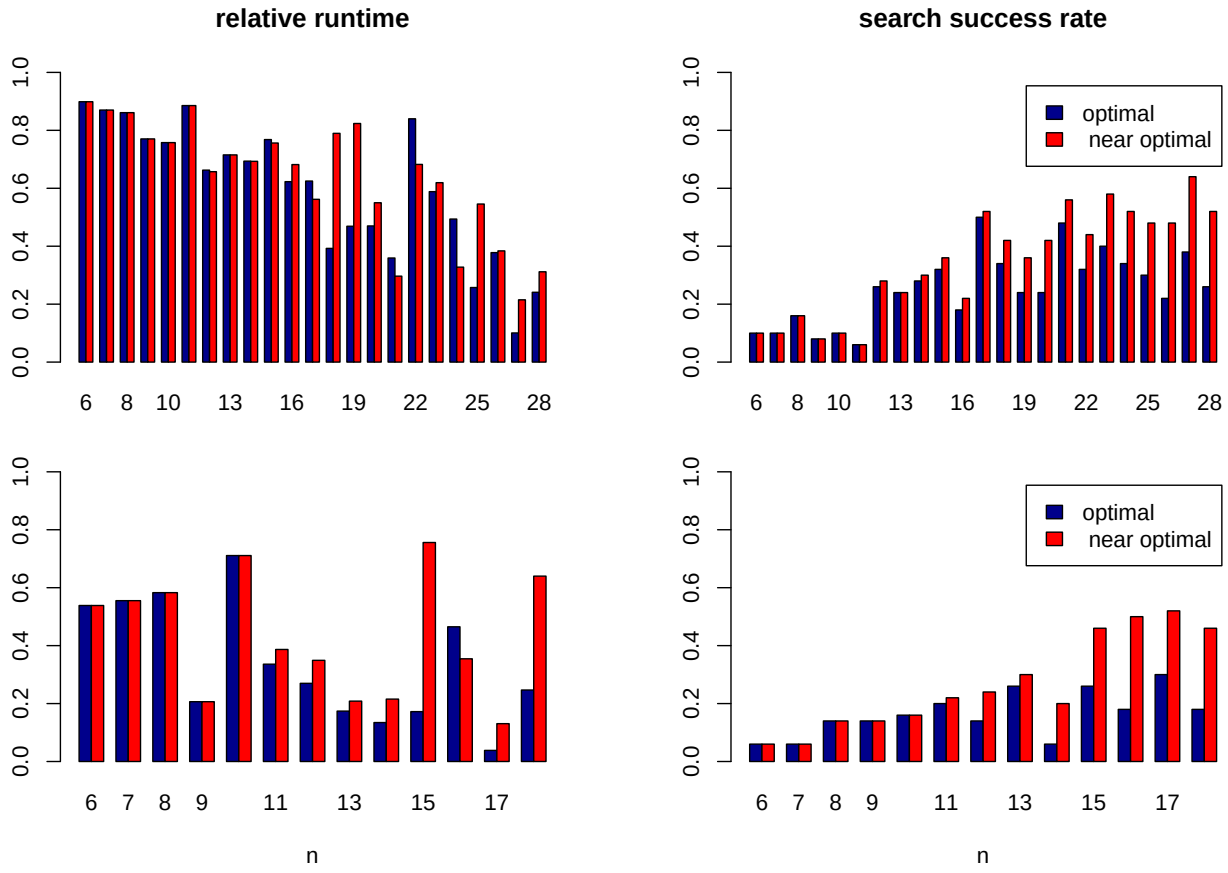

Figure 7.12: Structured random instances of sizes $n \times n \times 8$ (top row) and $n \times n \times 15$ (bottom row)

on uniform instances than structured ones. This is likely due to both the fact that the complete search performs much better on uniform instances (due to the large number of zeroes) and the LNS profiling showed much greater deviation on structured instances making it more difficult to *fix* variables to effective values.

Both the LNS search and the *Counter Model* search were implemented in COMET 3.0. The benchmarks were run on a MacBook Air with an Intel Core 2 Duo processor 2.13 Ghz with 4 GB of memory and a 256 GB SSD drive running MacOS X (10.6.7)

Chapter 8

Generic Lagrangian Relaxation

Lagrangian relaxation (introduced in section 3.6) is a classic optimization paradigm which is largely independent of how the model is expressed (and solved). It is heavily used in continuous optimization (example [51]) and in Mixed Integer Programming (example [36]). It had attracted some attention in constraint satisfaction problems and SAT in the late 1990s (e.g., [99, 19]) but has not been a topic of much research since then.

This chapter (and the original paper [37]) originated from an attempt to build model combinators for lagrangian relaxation that would apply to arbitrary optimization models. The design of these combinators required a systematic investigation of the semantics of Lagrangian relaxation over these models, which revealed some interesting modeling, computational, and implementation issues. It also raised the question about the potential benefits of Lagrangian relaxation for constraint programming, hybrid methods, and constraint-based local search, a topic which has been largely neglected in the constraint-programming community.

Generic lagrangian relaxation requires some new theoretical machinery which will be built up in this chapter culminating in a *generic lagrangian combinator*. The concept of *satisfiability degree* will be introduced, which provides an alternative to the notion of constraint violation used in constraint programming (e.g., [12]) and constraint-based local search [22, 58, 89]. Natural generalizations of traditional Lagrangian relaxation concepts, including the Lagrangian duals, subgradient optimization [36], surrogate subgradient optimization [103], and primal Lagrangian methods (e.g., [99, 19]) will be defined. These generalizations then makes it possible to design model combinators

for Lagrangian relaxation that apply to arbitrary models and algorithmic templates for Lagrangian methods that are independent of the solving technology.

Lagrangian relaxations of abstract models can then be concretized into a variety of optimization technologies (in particular, constraint programming, local search, or MIP). The resulting concrete optimization programs can be entrusted to algorithmic templates to solve *lagrangian dual* or use *lagrangian primal* methods.

8.1 Generalized Lagrangian Relaxation

This section explores how the traditional formulation of Lagrangian relaxations and Lagrangian duals can be systematically generalized.

8.1.1 Violation and Satisfiability Degrees

Generalized Lagrangian relaxations are based on the concepts of violation and satisfiability degrees. The violation degree of a constraint is a key concept in constraint programming (e.g., [12]) and constraint-based local search (e.g., [58, 89, 22]). The violation degree is constraint-dependent and exploits the constraint structure. Intuitively, the violation degree denotes how violated the constraint is.

Violation Degree The violation degree of constraint $c : \Re^n \rightarrow \mathit{Bool}$ is a function $\nu_c : \Re^n \rightarrow \Re^+$ such that

$$c(v_1, \ldots, v_n) \equiv \nu_c(v_1, \ldots, v_n) = 0.$$

In contrast, the satisfiability degree of a constraint captures both how much the constraint is violated and the constraint slackness when it is satisfied. It generalizes the Lagrangian relaxation typically used in mathematical programming. Intuitively, when the satisfiability degree is strictly positive, it denotes how much the constraint is violated; when it is negative, the constraint is satisfied and the satisfiability degree denotes the *slack* of the constraint. When the satisfiability degree is zero, the constraint is satisfied but tight.

Satisfiability Degree The satisfiability degree of constraint $c : \Re^n \rightarrow \mathit{Bool}$ is a function $\sigma_c : \Re^n \rightarrow \Re$ such that

$$c(v_1, \ldots, v_n) \equiv \sigma_c(v_1, \ldots, v_n) \leq 0.$$

Example ($\boldsymbol{A \cdot x \geq b}$) Consider the constraint $c(x)$ defined by $A \cdot x \geq b$. Its satisfiability degree is given by the function $\sigma_c(x) = b - A \cdot x$. Observe that, when $\sigma_c(v) \leq 0$, $A \cdot v \geq b$ and $c(v)$ is satisfied. When $\sigma_c(v) > 0$, $A \cdot v < b$ and $\sigma_c(v)$ represents how much the constraint is violated. When $\sigma_c(v) = 0$, the constraint is satisfied at equality. When $\sigma_c(v) < 0$, $\sigma_c(v)$ captures the slackness of the inequality.

Example (`disjunctive`$\boldsymbol{(s_1, d_1, s_2, d_2)}$) Constraint `disjunctive`$(s_1, d_1, s_2, d_2)$ holds if $s_1 + d_1 \leq s_2 \vee s_2 + d_2 \leq s_1$. Its degree of satisfiability is given by

$$\sigma_d(s_1, d_1, s_2, d_2) = \min(s_1 + d_1, s_2 + d_2) - \max(s_1, s_2).$$

When $\max(s_1, s_2) < \min(s_1 + d_1, s_2 + d_2)$, the two intervals $[s_1, s_1 + d_1]$ and $[s_2, s_2 + d_2]$ overlap, the disjunctive constraint is violated and $\sigma_d(s_1, d_1, s_2, d_2) > 0$. Similarly, if $\max(s_1, s_2) \geq \min(s_1 + d_1, s_2 + d_2)$, the two intervals are temporally separated and $|\sigma_d(s_1, d_1, s_2, d_2)|$ is the temporal slack separating the end of the first activity from the start of the second activity.

The following example illustrates that, for some constraints, the satisfiability degree reduces to the violation degree.

Example (`permutation`$\boldsymbol{(x_1, \cdots, x_n)}$) Constraint `permutation`$(x_1, \cdots, x_n)$ holds if $x_1, \cdots, x_n$ is a permutation of the values in interval $1..n$. Its degree of satisfiability is given by

$$\sigma_p(v_1, \cdots, v_n) = \sum_{j=1}^{n} \max\left(0, \left(\sum_{i=1}^{n} (x_i = j)\right) - 1\right).$$

When $\sigma_p(v_1, \cdots, v_n) > 0$, the constraint is violated. When $\sigma_p(v_1, \cdots, v_n) = 0$, each value is selected exactly once and the constraint is satisfied. However, $\sigma_p(v_1, \cdots, v_n)$ is never negative as all permutations are equally good: None satisfies the constraint more than the others.

Observe that the violation degree ν_c of a constraint c can always be defined in terms of its satisfiability degree σ_c by stating

$$\nu_c(x_1, \ldots, x_n) = \max(0, \sigma_c(x_1, \ldots, x_n)).$$

We make this assumption in the rest of this chapter, when comparing relaxations based on ν_c and σ_c.

8.1.2 Generalized Lagrangian Relaxations

This section considers Lagrangian relaxations based on violation and satisfiability degrees. It first defines the concepts of constraint softening and constraint relaxation.

Constraint Softening The softening of a constraint c over $\Re^n$ is a constraint $soft(c)$ over $\Re^{n+1}$ defined as

$$soft(c)(x_1, \cdots, x_n, y) \equiv y = \nu_c(x_1, \cdots, x_n).$$

Constraint Relaxation The relaxation of a constraint c over $\Re^n$ is a constraint $relax(c)$ over $\Re^{n+1}$ defined as

$$relax(c)(x_1, \cdots, x_n, y) \equiv y = \sigma_c(x_1, \cdots, x_n).$$

We are now in a position to define generalized and soft Lagrangian relaxations.

Generalized and Soft Lagrangian Relaxations Consider the optimization problem

$$\begin{array}{ll} P = \min_x & f(x) \\ \text{subject to} & \left\{ \begin{array}{ll} c_h(x) & (h \in H) \\ c_e(x) & (e \in E) \end{array} \right. \end{array}$$

where H and E denote, respectively, the index sets of hard and easy constraints. The generalized

Lagrangian relaxation of P for a set of Lagrangian multipliers $\lambda_h \geq 0$ is given by

$$\begin{array}{lcl} GLR(\lambda) & = & \min_x f(x) + \sum_{h \in H} \lambda_h \cdot \sigma_h \\ \text{subject to} & & \begin{cases} c_e(x) & (e \in E) \\ relax(c_h)(x, \sigma_h) & (h \in H) \end{cases} \end{array}$$

The *soft* Lagrangian relaxation of P for a set of Lagrangian multipliers $\lambda_h \geq 0$ is given by

$$\begin{array}{lcl} SLR(\lambda) & = & \min_x f(x) + \sum_{h \in H} \lambda_h \cdot \nu_h \\ \text{subject to} & & \begin{cases} c_e(x) & (e \in E) \\ soft(c_h)(x, \nu_h) & (h \in H) \end{cases} \end{array}$$

The definitions of the generalized and soft Lagrangian relaxations are compositional and constraint-driven. This makes it possible to define model combinators that systematically obtain a Lagrangian relaxation from a high-level model as discussed in Section 8.4. The following (direct) lemma establishes the soundness of the approach.

Lemma 8.1.1 (Relaxations) *Consider the optimization problem P defined above, an optimal solution x^* of P, and the generalized and soft relaxations $GLR(\lambda)$ and $SLR(\lambda)$ for a vector $\lambda \geq 0 \in \mathbb{R}^{|H|}$. Then, $GLR(\lambda)$ and $SLR(\lambda)$ are relaxations of P, i.e., $GLR(\lambda) \leq f(x^*)$ and $SLR(\lambda) \leq f(x^*)$.*

Proof Each feasible solution of P satisfies $\nu_h = 0$ and $\sigma_h \leq 0$ for all $h \in H$. Hence, in $SLR(\lambda)$ (resp. $GLR(\lambda)$), the objective value of a feasible solution is the same as (resp. no greater than) the objective value of a feasible solution in P. The results follows since the λ_h are nonnegative. ∎

The following (also direct) lemma shows that the soft relaxation is at least as strong as the generalized relaxation.

Lemma 8.1.2 (GLR versus SLR) *For any $\lambda \geq 0$, we have $GLR(\lambda) \leq SLR(\lambda)$.*

This lemma seems to suggest the use of $SLR(\lambda)$ instead of $GLR(\lambda)$ (which generalizes the traditional mathematical approach), since it is a stronger relaxation. Indeed, $SLR(\lambda)$ could be defined as

$$\begin{array}{lcl} SLR(\lambda) & = & \min_x f(x) + \sum_{h \in H} \lambda_h \cdot \nu_h \\ \text{subject to} & & \begin{cases} c_e(x) & (e \in E) \\ relax(c_h)(x, \sigma_h) & (h \in H) \\ \nu_h \geq 0 & (h \in H) \\ \nu_h \geq \sigma_h & (h \in H) \end{cases} \end{array}$$

which does not change the theoretical complexity of the relaxation if $GLR(\lambda)$ is a linear program or a mixed integer program. The experimental results in Section 8.5 will shed some light on this issue.

8.2 Generalized Lagrangian Duals

Lagrangian relaxation is often used to find tight dual bounds to optimization problems. The aim is thus to determine the set of Lagrangian multipliers λ that gives the strongest dual bound. This section focuses on the generalized Lagrangian relaxation but the results apply to the soft Lagrangian relaxation as well. The generalized Lagrangian dual can then be defined as

$$GLR^* = \max_{\lambda \geq 0} GLR(\lambda).$$

The generalized Lagrangian dual satisfies the following property.

Theorem 8.2.1 (Optimality Test) *Let $\hat{x}$ be an optimal solution to $GLR(\lambda)$ for some $\lambda \geq 0$ such that*

1. *$c_h(\hat{x})$ holds for all $h \in H$.*
2. *$\lambda_h \cdot \sigma_h = 0$ for all $h \in H$.*

Then, $\hat{x}$ is an optimal solution to P and $GLR^* = GLR(\lambda)$.

```
1  function SubgradientSolve(GLR(λ), Z_UB)
2          π = 2
3          k = 0
4          λ^0 = 0⃗
5          Z_best = −∞
6          noImproveCount = 0
7          do
8                  x^{k+1} = solve(GLR(λ^k))
9                  Z^{k+1} = f(x^{k+1}) + Σ_{h∈H} λ^k_h · σ_h(x^{k+1})
10                 Δ^{k+1} = π(Z_UB − Z^{k+1})/||σ(x^{k+1})||^2
11                 forall (h ∈ H)  λ^{k+1}_h = max(0, λ^k_h + Δ^{k+1} * σ_h(x^{k+1}))
12                 if Z^{k+1} > Z_best
13                         Z_best = Z^{k+1}
14                         noImproveCount = 0
15                 else noImproveCount = noImproveCount + 1
16                 if noImproveCount > 30
17                         π = π/2
18                         noImproveCount = 0
19                 k = k + 1
20         while the termination criterion is not met;
21         return Z_best
```

Figure 8.1: The *subgradient* Algorithm Template.

Proof By condition (1) and the definition of $GLR(\lambda)$, $\hat{x}$ is a feasible solution. Moreover, by condition (2), $GLR(\lambda) = f(\hat{x}) + \sum_{h \in H} \lambda_h \cdot \sigma_h = f(\hat{x})$. Since $f(\hat{x})$ is both a lower and an upper bound, the result follows. ∎

Note that, for *SLR*, condition (1) implies condition (2).

Subgradient Optimization

The generalized Lagrangian dual can be rewritten explicitly as

$$\begin{aligned} &\max_{\lambda \geq 0} w \\ &\text{subject to} \\ &\quad w \leq f(x) + \lambda^T \cdot \sigma_h(x) \qquad \forall x, e \in E : c_e(x). \end{aligned}$$

This formulation has exponentially many constraints but it can be solved by a subgradient method which iterates two steps

$$\begin{aligned} x^{k+1} &= solve(GLR(\lambda^k)) \\ \lambda_h^{k+1} &= \max\left(0, \lambda_h^k + \Delta^{k+1}\ \sigma_h(x^{k+1})\right) \quad (h \in H) \end{aligned}$$

where Δ^k is the step size at iteration k. What remains to determine is the initial value of the

multipliers and the step size at each iteration. The algorithmic schema for subgradient optimization is depicted in Figure 8.1 and is independent of the model and the solving technology. Its input is a parametric Lagrangian model $GLR(\lambda)$ and an initial upper bound to the original problem P. The algorithmic template also uses an agility parameter π used to compute the step sizes. The subgradient optimization sets the initial multipliers λ^0 to 0. Lines 8–19 repeatedly solves the parametric model with the current multipliers λ^k and store its solution in x^{k+1} and its objective value in Z^{k+1} (lines 8–9). Lines 10 computes the step function Δ^{k+1} used on line 11 to compute the next multipliers λ^{k+1}. Lines 12–19 record the current best objective and update the agility parameter π when there is no improvement over some time. Observe that the template does not prescribe any technology for solving $GLR(\lambda^k)$.

Generalized Surrogate Optimization

The surrogate gradient method was introduced in [103] and refined in [85] to solve a Lagrangian dual featuring loosely coupled subproblems. By relaxing the coupling constraints, the subproblems can then be optimized independently.

Consider the following simple IP minimization problem:

$$
\begin{aligned}
Z = \texttt{min}\ & \textstyle\sum_{i=1}^{4} x_i \\
\texttt{s.t.}\ & \begin{cases} x_1 + x_2 & & \geq b_1 \\ & x_3 + x_4 & \geq b_2 \\ x_1 + & x_3 + x_4 & \geq b_3 \end{cases}
\end{aligned}
$$

Relaxing the coupling constraint produces the Lagrangian dual:

$$
\begin{aligned}
Z_{LD} = \texttt{min}\ & \textstyle\sum_{i=1}^{4} x_i + \lambda(b_3 - (x_1 + x_3 + x_4)) \\
\texttt{s.t.}\ & \begin{cases} x_1 + x_2 & & \geq b_1 \\ & x_3 + x_4 & \geq b_2 \end{cases}
\end{aligned}
$$

The objective function of Z_{LD} can be simplified algebraically in order to obtain two separable subproblems:

$$min[x_1(1-\lambda) + x_2] + [(x_3 + x_4)(1-\lambda)]$$

```
1  function SurrogateSolve(GLR(λ), Z_UB, {V_1, ..., V_k})
2          k = 0
3          λ^0 = 0⃗
4          Z_best = −∞
5          noImproveCount = 0
6          x^0 = Solve(GLR(λ^0))
7          do
8                  Z^k = f(x^k) + Σ_{h∈H} λ_h^k σ_h(x^k)
9                  Δ^k = (Z_UB − Z^k)/||σ(x^k)||^2
10                 forall (h ∈ H)  λ_h^{k+1} = max(0, λ_h^k + Δ^k * σ_h(x^k))
11                 if Z^k > Z_best
12                         Z_best = Z^k
13                         noImproveCount = 0
14                 else noImproveCount = noImproveCount + 1
15                 select  i ∈ 1..k
16                 y = Solve(GLR(λ^{k+1}, x^k, V_i))
17                 obj = f(y) + Σ_{h∈H} λ_h^{k+1} σ_h(y)
18                 if obj < z^k
19                     x^{k+1} = y
20                 else x^{k+1} = x^k
21                 k = k + 1
22         while the termination criterion is not met;
23         return Z_best
```

Figure 8.2: The *Surrogate Subgradient* Algorithm Template.

Such rewritings are not always possible when using arbitrary models featuring global constraints. But the surrogate subgradient algorithm can be generalized to arbitrary models by using ideas from large neighborhood search. At each iteration, a subproblem can be chosen and all the variables not appearing in this subproblem are fixed to their values in the incumbent solution. A subproblem $GLR(\lambda, \hat{x}, V)$, where $\hat{x}$ is an incumbent solution and V is the set of variables associated with one of the subproblems, can be defined as

$$GLR(\lambda, \hat{x}, V) \quad = \quad \min_x f(x) + \sum_{h \in H} \lambda_h \ \sigma_h(x)$$

$$\text{subject to} \quad \begin{cases} c_e(x) & (e \in E) \\ x_i = \hat{x}_i & (i \notin V) \end{cases}$$

With this idea in mind, the generalized surrogate gradient template is presented in Figure 8.2. It receives as inputs the parametric model and the set of variables appearing in each subproblem. Observe that line 6 solves the initial model entirely before starting the subproblem optimization. Once again, the template does not prescribe any technology for solving $GLR(\lambda^k, \hat{x}, V)$.

8.3 Generalized Lagrangian Primal Methods

Primal Lagrangian methods are ubiquitous in continuous optimization. In the late 1990s, some of their main concepts were elegantly transferred to discrete optimization [80, 99]. Focusing on violation degrees, the resulting Lagrangian primal methods (SPLR) can be viewed as the iteration of two steps:

$$\begin{aligned} x^{k+1} &= \text{argmin}_{x \in \mathcal{N}(x^k)} SLR(\lambda^k, x^k) \\ \lambda^{k+1} &= \lambda^k + \nu(x^{k+1}) \end{aligned}$$

where $\mathcal{N}(x)$ is the neighborhood around x, i.e., a set of points satisfying the easy constraint and including x, and $SLR(\lambda, x) = f(x) + \lambda\nu(x)$. Such primal Lagrangian methods thus descend in the x-space and ascend in the λ-space. Such primal Lagrangian methods were applied to SAT [80] and constraint satisfaction problems [19], with neighborhoods changing the value of one variable. However, they have not attracted much attention in the constraint-programming community since then. It is useful to state the main theoretical results from [99], since they shed some light on the search algorithm.

Discrete Saddle Point A pair (λ^*, x^*) is a discrete saddle point of SLR if $SLR(\lambda, x^*) \leq SLR(\lambda^*, x^*) \leq SLR(\lambda^*, x)$ for all λ and $x \in \mathcal{N}(x^*)$.

The left condition in the definition can be shown to be equivalent to $\nu(x^*) = 0$. The following theorem is a direct adaptation to our setting of the main results in [99].

Theorem 8.3.1 (Saddle Point Theorem) *Point x^* is a local minimum to the original problem P if and only if there exists $\lambda^* \geq 0$ such that (λ^*, x^*) is a discrete saddle point. Moreover, (λ^*, x^*) is a saddle point if and only if $x^* = argmin_{x \in \mathcal{N}(x^*)} SLR(\lambda^*, x^*)$ and $\nu(x^*) = 0$.*

Observe also that if (λ^*, x^*) is a saddle point, so is (λ, x^*) for $\lambda \geq \lambda^*$ [99]. Hence, in theory, there is no need to decrease the Lagrangian multipliers when searching for a saddle point. It is thus unclear whether the satisfiability degree is useful in primal Lagrangian methods.

In contrast to earlier work, this work studies whether primal Lagrangian methods can provide a simple, systematic, and principled way of boosting existing search methods, such as tabu search or large neighborhood search, when applied to high-level models. In other words, the neighborhood $\mathcal{N}$

in these primal Lagrangian methods is very large and defined by a neighborhood search technique over a high-level model.

8.4 Practical Implementation

The earlier sections defined a general framework for applying Lagrangian relaxation to high-level models. This section describes how this generality is supported in Objective-CP [94]. Intuitively, the implementation starts with a high-level model which is then relaxed by replacing the hard constraints with their relaxation and adding a new term in the objective function to capture the weighted sum of violations or satisfiability degrees. The hard constraints are identified either by users or automatically by a partitioning algorithms. The resulting Lagrangian model is then concretized into an optimization program, which can be a MIP solver, a constraint-programming solver, or a constraint-based local search. The concrete optimization program is then embedded in an algorithmic template (a runnable in Objective-CP's terminology [39], e.g., a surrogate dual or a primal Lagrangian methods. We now illustrate this methodology on a few code snippets. Consider the excerpt:

```
1 id<ORModel>  P = [ORFactory createModel];
2 ...
3 id<ORIdArray>  H  = ... // array of hard constraints in P
4 id<ORModel>    L  = [ORFactory lagrangianRelax: P relaxingConstraints: H];
5 id<ORProgram>  O  = [ORFactory createMIPProgram: L];
6 id<ORRunnable> r  = [ORFactory subgradient: O];
7 [r run];
```

The code fragment starts by declaring a model P on line 1. Line 3 stores the set of constraints deemed hard in P in array H. Line 4 creates a parametric model L representing $GLR_P(\lambda)$. Line 5 concretizes $GLR_P(\lambda)$ into a parametric MIP program O, which is solved using a subgradient template in Lines 6–7. To switch to a `CP` solver, it suffices to change line 5 into

```
1 id<ORProgram> O = [ORFactory createCPProgram: L];
```

Similarly, to use violation degrees rather than satisfiability degrees, it is sufficient to edit line 4 to

read

```
1 id<ORModel>    L = [ORFactory lagrangianRelax: P softeningConstraints: H];
```

Observe that, following [39], Objective-CP records the fact that L is a relaxation of P and the runnable produces several products in agreement with a relaxation specification, including a stream of lower bounds. It can thus be composed naturally with a primal algorithm.

Consider now the application of a surrogate optimization scheme:

```
1 id<ORModel>      P = [ORFactory createModel];
2 ...
3 id<ORIdArray>    H = ... // array of hard constraints in P
4 id<ORPartition> Vs = [ORFactory autoPartition: P           accordingTo: H];
5 id<ORModel>      S = [ORFactory lagrangianRelax: P relaxingConstraints: H];
6 id<ORProgram>    O = [ORFactory createMIPProgram: S];
7 id<ORRunnable>   r = [ORFactory surrogate: O splitWith: Vs];
8 [r run];
```

Line 4 computes a partition of the variables in P from the hard constraints. Line 5 creates the Lagrangian relaxation of P with respect to H and line 6 creates a MIP program that is then used by a surrogate runnable in line 7. The partition in line 4 is the argument $\{V_1, \cdots, V_k\}$ appearing in the template in Figure 8.2.

Models with a natural decomposable or 'block' structure are often difficult to decompose by hand, particularly for larger problems. Hence, it is useful to have the ability to automatically partition a problem based on sets of *coupling* constraints. Objective-CP makes use of a hyper-graph clustering algorithm [49] to provide an automatic decomposition. The variables of a model become nodes and each constraint defines an hyper-edge connecting it variables. The algorithm recursively clusters variables into disjoint sets until the maximal decomposition is achieved.

8.5 Case Studies

This section reports experimental results highlighting the concepts described in this chapter. The goal is not to present state-of-the-art results on specific problems but to make the case that La-

grangian relaxation could play a larger role in the constraint-programming community. In addition, the experiments present some interesting perspectives on some design choices in Lagrangian methods. All experimental results are obtained using OBJECTIVE-CP [94] unless specified otherwise. Mixed-Integer programs are solved using Gurobi 5.6.

8.5.1 Graph Coloring

Graph coloring (see section 1.2.2) aims at minimizing the number of colors necessary to color a graph so that no two adjacent vertices have the same color. The following is a typical CP formulation of the problem:

$$Z_{CP} = \texttt{min}\ \ m$$
$$\text{subject to} \begin{cases} v_i \leq m, & i \in 1..|V| \\ v_i \neq v_j, & (i,j) \in E \\ v_i \in 1..|V|, & i \in 1..|V| \\ m \in 1..|V| \end{cases}$$

Here, V is the set of vertices, E the set of edges, m a decision variable for the number of colors used and $\{v_i\}_{i \in 1..|V|}$ are decision variables representing the color assigned to the i-th vertex of V. In the Lagrangian relaxation, E is partitioned into a 'hard' and 'easy' edge set $E = E_e \cup E_h$, relaxing E_h. The experiments also use a MIP formulation automatically obtained from the above model by a linearization transformation. The OBJECTIVE-CP linearization performs a *binarization* of the variables $\{v_i\}_{i \in 1..|V|}$ over their domains and transforms the (non-linear) disequality constraints into sets of inequalities.

The experiments consider three dual methods (GLR(MIP), SLR(MIP), and SLR(CP)), as well as two primal methods (MIP, CP). The methods are evaluated on randomly generated instances[1] which are built around collections of vertex cliques connected via *coupling edges*. More precisely, the vertex set is first partitioned into randomly sized cliques. Then a subset of vertices are chosen at random and *coupling edges* between these vertices are added. These instances are generated based on three parameters: *number of vertices* (nbv), *number of cliques* (nbc), *number of coupled*

[1]Python script for generating instances: `http://bit.ly/1jDCgJq`

Instances	Dual											Primal				
	GLR(MIP)				SLR(MIP)				SLR(CP)			MIP			CP	
	time	lb	ub	itr	time	lb	ub	itr	time	lb	itr	time	lb	up	time	ub
20-3-39	0.08	11*	11*	1	0.07	11*	11*	1	0.44	11*	10.95	0.06	11*	11*	**0.01**	11*
120-25-188	300	9	9	173	3.4	9*	9*	2	30.25	9*	8.35	**1.52**	9*	9*	1.98	9*
160-2-846	300	55	160	1	300	55	160	1	300	105	104.5	300	55	160	**0.07**	108*
160-30-187	300	9	9	103	6.77	9*	9*	2	0.11	9*	4.35	2.5	9*	9*	**0.03**	9*
80-10-176	300	13	13	266	2.15	13*	13*	2	8.25	13*	13.3	0.90	13*	13*	**0.02**	13*
200-10-281	300	30	30	30	23.48	30*	30*	2	12.14	30*	30.0	12.03	30*	30*	**11.03**	30*
120-3-465	300	53	53	33	300	53*	53*	34	37.25	53*	51.15	10.0	53*	53*	**0.05**	53*
160-4-498	300	62	62	16	40.4	62*	62*	2	54.04	62*	61.0	20.01	62*	62*	**6.92**	62*
120-5-938	300	34	34	49	300	34	34	54	300	34	33.5	**9.48**	35*	35*	25.94	35*
200-20-201	300	16	16	47	12.8	16*	16*	2	3.58	16*	16.15	5.77	16*	16*	**0.03**	16*
180-5-873	300	51	51	11	176.85	51*	51*	2	300	51*	49.5	**23.29**	51*	51*	37.04	51*
100-2-910	12.5	71*	71*	1	12.40	71*	71*	1	300	69	67.5	12.52	71*	71*	**0.03**	71*
150-5-1803	-	-	-	-	-	-	-	-	300	41	39.5	26.8	46*	46*	**11.11**	46*
140-12-1137	-	-	-	-	-	-	-	-	300	19	10.0	**12.04**	20*	20*	55.7	20*

Table 8.1: Experimental Results on *Graph Coloring*.

vertices (nbcv). In Figure 8.1, instances are referred to in the following format: 'nbv-nbc-nbcv'. The *relaxed edges* E_h used in GLR(MIP), SLR(MIP), and SLR(CP) are a subset of the *coupling edges*. The problem is first decomposed into independent sets (cliques in this case) using a standard hyper-graph partitioning algorithm. Edges which do not have a vertex in the maximal clique are relaxed. Initial upper bounds provided to the dual problem were about twice the optimal value.

Table 8.1 describes the results on 15 instances and it reports the runtime and the bounds produced by the various algorithms. Dual algorithms only report a lower bound *lb* while the MIP produces both lower and an upper bounds, and the CP program produces an upper bound only. Dual algorithms may terminate because of a timeout or because the step size is too small in which case the dual solution is typically primal-infeasible. Theorem 1 specifies when the Lagrangian dual produces an optimal solution. The table also reports the number of Lagrangian iterations. Bold entries correspond to the fastest implementation, while *starred* entries indicate whether an optimal solution was found and proved optimal. A timeout of 300 seconds is used throughout.

The results bring some interesting conclusions. The dual MIP approaches perform poorly on these benchmarks and are strongly dominated by the primal MIP. **The dual CP is better than the dual MIP approaches but is typically bettered by the Primal CP formulation. The primal CP approach is the most effective approach on a number of benchmarks but is sometimes dominated by the primal MIP. These results seem to indicate that it would be valuable to investigate combinations of Lagrangian relaxation and constraint programming systematically on more applications. Note

Inst	GLR			SLR 5s			SLR 10s			SLR		
	time	itr	bnd	time	itr	bnd	time	itr	bnd	time	itr	bnd
inst 1	243.8	154	155.8	900	13	164.0	900	15	163.2	900	7	161.6
inst 2	900	109	149.2	900	12	153.1	900	12	155.3	900	9	153.5

Table 8.2: Experiments on *Set Covering* problems.

that the absence of lower-bounds and the large number of symmetries is detrimental to SLR(CP) which explores alternative selections of violated colorings.

8.5.2 GLR versus SLR

Lemma 2 indicated that SLR is a stronger relaxation than GLR, although GLR is the traditional Lagrangian relaxation in mathematical programming. Experimental results on graph coloring indicated that $SLR(MIP)$ systematically outperforms $GLR(MIP)$ on these instances and performs significantly fewer iterations. This section aims at confirming these results on set-covering instances. The results are based on random instances generated in separable blocks of random sizes which are extended with *coupling constraints*. The instances consider 1000 elements and 400 sets partitioned into 10 separable blocks and 250 coupling constraints (which are relaxed). The results for two representative instances are presented in Figure 8.3 and Table 8.2. The results again indicate that GLR and SLR behave very differently. SLR tends to have longer iterations but make larger jumps, while GLR features relatively rapid iterations but makes much less progress per iteration. It is also possible to speed-up SLR substantially by using a time limit. SLR then returns its best lower bound at the time limit when an improved lower bound has been found; otherwise, it continues until such a lower bound is found or the search is complete. Figure 8.3 shows the bound quality of GLR, SLR, and the time-limited SLR with limits of 5 seconds (SLR 5s) and 10 seconds (SLR 10s). A 15 minute (900s) time out is used.

The results on these set-covering instances shed some light on the respective strengths of GLR and SLR. On the first instance, GLR terminates because the step size has become too small (due to lack of bound improvement). Once again, SLR approaches dominate GLR on these instances. Results on instance 2 is also revealing in that the smaller time limit gives better bounds early on but eventually falls behind and produces poorer bounds. Once again, these results seem to indicate

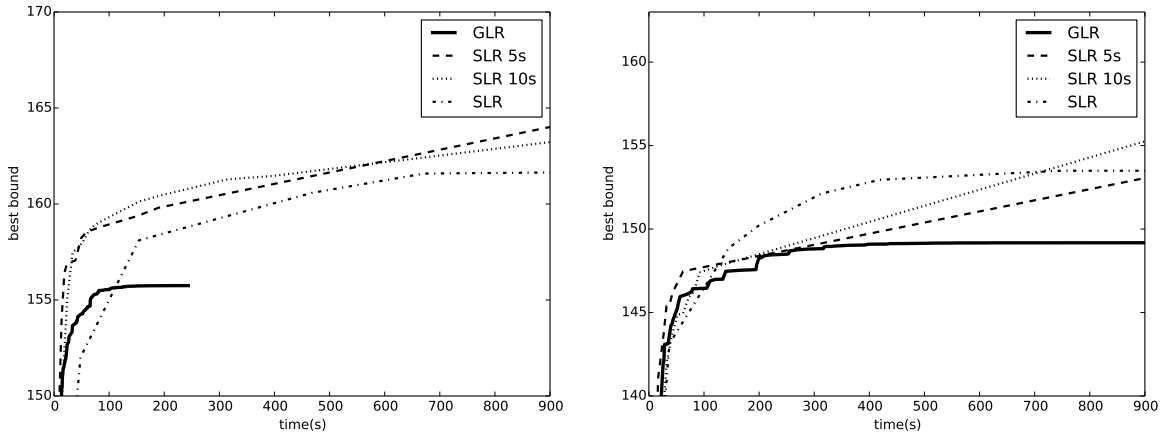

Figure 8.3: Bound quality over time, GLR vs SLR.

that concepts from constraint programming, i.e., the degree of violations, could bring benefits into traditional Lagrangian dual methods.

8.5.3 Primal Lagrangian Tabu Search

This section describes the application of Lagrangian primal methods to boost the performance of a tabu-search algorithm. The tabu search is used to explore the neighborhood in the SPLR method; Upon completion, the Lagrangian multipliers are updated using the violation degrees and the tabu-search algorithm is restarted. The experiments are performed on the hardest instances of the progressive party problem using the model and the tabu search presented in [89] but with no restarting component and no manual tuning of the constraint weights. The progressive party problem features a variety of global constraints (e.g., *alldifferent* and *packing* constraints). It is thus fundamentally different from the benchmarks typically used to demonstrate the weighting schemes which uses SAT or binary CSPs, or binary constraints (e.g., [80, 19, 79]).

Figure 8.4 reports the experimental results. SPLR is a primal Lagrangian method using the tabu search (with no restart) to explore the neighborhood $\mathcal{N}$. SPLR updates the weights after n^2 iterations of the tabu search, where n is the number of variables. Tabu is the tabu search with restarts, i.e., the control case. W-Tabu is the Comet implementation of the tabu search with hand-chosen constraint weights and restarts (running on a slightly faster processor). All

Algorithm	C	P	$\%S$	$m(I)$	$M(I)$	$\mu(I)$	$\sigma(I)$	$m(T)$	$M(T)$	$\mu(T)$	$\sigma(T)$
SPLR	3	9	98	63019	1000000	373515.82	233297.49	4.68	73.95	26.36	16.65
	4	9	94	73319	1000000	386478.46	246292.98	5.14	73.01	27.59	17.90
	6	7	94	40010	1000000	312445.44	280601.45	3.24	76.61	23.26	21.11
Tabu	3	9	30	22729	1000000	816440.38	311171.57	1.855	86.542	59.46	23.15
	4	9	48	17692	1000000	767466.30	313399,19	1.40	82.36	55.59	23.15
	6	7	64	130117	1000000	733028.38	285338.94	8.699	66.11	47.55	18.56
W-Tabu	3	9	96	23645	1000000	245331.96	249549.41	4.41	164.44	40.10	40.14
	4	9	94	47962	1000000	363842.50	273432.48	8.35	166.90	59.55	44.44
	6	7	88	19314	1000000	379072.73	328393.13	3.24	152.37	56.92	48.91

Figure 8.4: Experimental Results for SPLR on the Progressive Party Problem.

algorithms have a limit of 1,000,000 iterations and were executed 50 times on each instance. The table reports the configuration C, the number of periods P, the success percentage $\%S$ and the minimum, maximum, average, and standard deviation for the number of iterations and CPU time.

The results shows that SPLR significantly outperforms the tabu search in speed and success rate on these instances, indicating that Lagrangian primal methods may provide a simple, systematic, and principled way to boost the performance of meta-heuristics and complex search procedures. Moreover, SPLR exhibits a performance similar to W-Tabu on instances (3,9) and (4,9) and outperforms it slightly on instance (6,7). This indicates that primal Lagrangian methods may find proper multipliers quickly (the theory indicates that such multipliers exist but not how fast they can be identified).

It is also interesting to report some additional insights on SPLR. Indeed, experimental results show that using the satisfiability is counter-productive in this setting, the search rarely converging to a feasible solution. Moreover, using a restarting component is also not productive, which is a surprise given the importance on restarts for tabu search on this benchmark. The Lagrangian multipliers are very effective in driving the search out of local minima on this problems.

Summary: The experimental results show the versatility of Lagrangian relaxation for a variety of solver technologies and models. In particular, they show that

- The Lagrangian dual coupled with constraint programming is an effective method for some classes of graph coloring problems.
- The concept of violation degree is valuable to improve the quality and performance of the Lagrangian dual when solved with MIP solvers. It is not clear however whether satisfiability

degrees can be valuable for constraint programming or local search.

- Primal Lagrangian methods may systematically boost the performance and solution quality of meta-heuristics in a principled way.

Overall, these results tend to indicate that Lagrangian methods could play a much more significant role in constraint programming and large neighborhood search and that further synergies between constraint programming and mathematical programming should be explored.

Chapter 9

Implementation

9.1 Interfaces for Runnables & Combinators

All of the machinery presented thus far has been implemented as part of the Objective-CP Optimization Library. The library provides protocols for the major concepts introduces above including *abstract models* (ORModel), *model signatures* (ORSignature, Figure 9.1), *runnables* (ORRunnable, Figure 9.2) and *combinators* (ORCombinator, Figure 9.3).

```
1 @protocol ORSignature<NSObject>
2 -(ORBool) matches: (id<ORSignature>)sig;
3 -(ORBool) isComplete;
4 -(ORBool) providesUpperBound;
5 -(ORBool) providesUpperBoundStream;
6 -(ORBool) providesLowerBound;
7 -(ORBool) providesLowerBoundStream;
8 -(ORBool) providesLowerBoundPool;
9 -(ORBool) providesUpperBoundPool;
10 -(ORBool) providesSolutionStream;
11 -(ORBool) providesColumn;
12 -(ORBool) providesConstraint;
13 -(ORBool) providesConstraintSet;
14 -(ORBool) acceptsUpperBound;
15 -(ORBool) acceptsUpperBoundStream;
16 -(ORBool) acceptsLowerBound;
17 -(ORBool) acceptsLowerBoundStream;
18 -(ORBool) acceptsLowerBoundPool;
19 -(ORBool) acceptsUpperBoundPool;
20 -(ORBool) acceptsSolutionStream;
21 -(ORBool) acceptsColumn;
22 -(ORBool) acceptsConstraint;
23 -(ORBool) acceptsConstraintSet;
24 @end
```

Figure 9.1: Protocol for ORSignature

Figure 9.1 shows the protocol for *ORSignature* which has methods to check if the search is complete (line 3), check if the signature matches some other signature (line 2) and query input and output capabilities (lines 4-23). The 'match' method does not check for equality, but rather it verifies that the target signature contains all the capabilities present in the input signature. This is useful for checking preconditions on a *runnable*'s signature and verifying that it meets some minimum requirements set by a combinator.

```
1 @protocol ORRunnable<NSObject>
2 -(id<ORModel>) model;
3 -(id<ORSignature>) signature;
4 -(void) start;
5 -(void) run;
6 -(void) setTimeLimit: (ORDouble) secs;
7 -(ORDouble) bestBound;
8 -(id<ORSolution>) bestSolution;
9 -(void)cancelSearch;
10 @end
11
12 @interface ORFactory(ORRunnable)
13 +(id<ORRunnable>) CPRunnable: (id<ORModel>)m;
14 +(id<ORRunnable>) CPRunnable: (id<ORModel>)m numThreads: (ORInt)nth;
15 +(id<ORRunnable>) CPRunnable: (id<ORModel>)m
16         solve: (void(^)(id<CPCommonProgram>))body;
17 +(id<ORRunnable>) CPRunnable: (id<ORModel>)m numThreads: (ORInt)nth
18         solve: (void(^)(id<CPCommonProgram>))body;
19 +(id<ORRunnable>) LPRunnable: (id<ORModel>)m;
20 +(id<ORRunnable>) MIPRunnable: (id<ORModel>)m;
21 +(id<ORRunnable>) MIPRunnable: (id<ORModel>)m numThreads: (ORInt)nth;
22 @end
```

Figure 9.2: Protocol for ORRunnable

Figure 9.2 shows the protocol for an *ORRunnable* on lines 1-11. A *runnable* must have an associated abstract model (line 2) and a signature (line 3) as well as methods for running the solver, querying the best solution, setting time limits on the search and canceling the search. Lines 13-23 define factory methods for creating 'primitive' runnables for CP, MIP and LP with an allocated number of threads.

```
1 @protocol ORCombinator<NSObject>
2 -(BOOL) isCompatible: (NSArray*)runnables;
3 -(id<ORRunnable>) apply: (NSArray*)runnables;
4 @end
```

Figure 9.3: Protocol for ORCombinator

Figure 9.3 shows the *ORCombinator* protocol. The combinator has only two methods, one to check its compatibility with an array of runnables and one to compose runnables.

```
1 id<ORModel> ATSPModel =   // Def. of ATSP Model
2 id<ORModel> LinearModel = [ORFactory linearizeModel:  ATSPModel];
3 id<ORModel> ContinuousModel = [ORFactory continuousRelax: LinearModel];
4 id<ORRunnable> r0 = [ORFactory CPRunnable: ATSPModel];
5 id<ORRunnable> r1 = [ORFactory IPRunnable: LinearModel];
6 if<ORRunnable> r2 = [ORFactory LPRunnable: ContinuousModel];
7 id<ORRunnable> complete = [ORCombinator completeParallel:cp0 with: ip1];
8 id<ORRunnable> relaxed  = [ORCombinator relaxParallel:complete with: lp2];
9 [relaxed run];
```

Figure 9.4: Running two encodings of the Asymmetric Traveling Salesman Problem in parallel.

```
1 BOOL^(id<ORRunnable> r1, id<ORRunnable> r2) {
2         return ([ORFactory isRelaxation: r2 of: r1] &&
3                 [r1.signature acceptsLowerBoundsStream] &&
4                 [r1.signature producesSolutionStream] &&
5                 [r2.signature producesLowerBoundStream]);
6 }
```

Figure 9.5: Precondition closure for ORRelaxedParallelCombinator.

The interesting aspect of the implementation is how signatures and preconditions are interpreted and how the *pipes* is derived from model signatures. Consider the following example. Suppose, we have defined an Asymmetric Traveling Salesman Problem (ATSP) as an abstract model and we wish to run a CP concretization of the abstract model in parallel with both an IP linear encoding of the problem and an LP relaxation of the linear encoding. This is done simply with the Objective-CP code in Figure 9.4.

The preconditions for a combinator is implemented as a Boolean valued first-order function accepting the child runnables as input. Figure 9.5 shows the Objective-CP code for the precondition on `ORRelaxedParallelCombinator`. The method is simply comprised of a Boolean expression which first verifies that the models are related by *relaxation* and then verifies that r_1 accepts lower bounds and produces solutions while r_2 produces lower bounds.

9.2 Communication between Runnables

Pipes (both internal and external) rely heavily on Objective-CP's `ORInformer` architecture, which provides a thread-safe *producer-consumer* based event system. Figure 9.6 shows some of the protocols for the `ORInformer` architecture. *ORRunnables* have an `ORInformer` object for

```
1 @protocol ORInformer<NSObject>
2 -(void) whenNotifiedDo: (id) closure;
3 -(void) wheneverNotifiedDo: (id) closure;
4 -(void) sleepUntilNotified;
5 @end
6
7 @protocol ORVoidInformer<ORInformer>
8 -(void) notify;
9 @end
10
11 @protocol ORIntInformer<ORInformer>
12 -(void) notifyWith:(int)a0;
13 @end
14
15 @protocol ORDoubleInformer<ORInformer>
16 -(void) notifyWithFloat:(double)a0;
17 @end
18
19 @protocol ORSolutionInformer<ORInformer>
20 -(void) notifyWithSolution: (id<ORSolution>)s;
21 @end
22
23 @protocol ORConstraintInformer<ORInformer>
24 -(void) notifyWithConstraint: (id<ORConstraint>)c;
25 @end
26
27 @protocol ORDoubleArrayInformer <ORInformer>
28 -(void) notifyWithDoubleArray: (id<ORDoubleArray>)arr;
29 @end
30
31 @protocol ORConstraintSetInformer <ORInformer>
32 -(void) notifyWithConstraintSet: (id<ORConstraintSet>)s;
33 @end
```

Figure 9.6: Protocols for the `ORInformer` architecture.

each of the *products* they are capable of producing. As an *ORRunnable* executes, it makes calls to 'notifyWithSolution' (line 20) as it finds a solution, 'notifyWith' (line 12) as it discovers a new bound or 'notifyWithConstraintSet' (line 32) for communicating a set of cuts. Other *ORRunnables* may be connected to these events by setting up callbacks. Figure 9.8 shows a callback which will execute a closure (receiving a solution) whenever the informer is triggered by a 'notifyWithSolution' call. These callbacks may be set up by calling one of the methods on lines 2-3.

Leveraging the `ORInformer` architecture, a runnable will provide a set of informers that matches the pipes described in its signature. Figure 9.7 show the function in Objective-CP for setting up the internal connections for the `ORCompleteParallelCombinator`. The figure shows that the child runnables (r_1 and r_2) are interconnected as producers and consumers of solution streams. Other connections such as upper bound and lower bound streams are made in this function, but are not shown for brevity. At the bottom of the function, the parent is connected as

```
1 void internal(id<ORRunnable> parent,id<ORRunnable> r1, id<ORRunnable> r2) {
2   if([[r1 signature] providesSolutionStream] &&
3      [[r2 signature] acceptsSolutionStream])
4        [r1 addSolutionStreamConsumer: r1];
5   if([[r2 signature] providesSolutionStream] &&
6      [[r1 signature] acceptsSolutionStream])
7        [r2 addSolutionStreamConsumer: r1];
8     ...
9     // Connect the parent as a listener on child solution streams
10    if([[r1 signature] providesSolutionStream])
11        [r1 addSolutionStreamConsumer: parent];
12    if([[r2 signature] providesSolutionStream])
13        [r2 addSolutionStreamConsumer: parent];
14 }
```

Figure 9.7: Internal pipe closure for ORCompleteParallelCombinator.

```
1 [[self solutionStreamInformer] wheneverNotifiedDo: ^void(id<ORSolution> s) {
2   for(id<ORSolutionStreamConsumer> c in _solutionStreamConsumers)
3     [[c solutionStreamInformer] notifyWith: s];
4 }];
```

Figure 9.8: Output pipe achieved through a multicast whenever a child solution is captured by the parent.

a listener to the solution streams of its children. This allows the parent to capture the streams of solutions coming from its children and make them available via a multicast as part of the parent's output pipe (shown in Figure 9.8). Note that the *ORInformer* architecture existed in Objective-CP prior to the work on this thesis, though thesis specific work was done to expand *ORInformer* to accommodate additional types of communication (columns and pools of constraints for example).

A very rich system of communication can be automatically generated by relying on the *ORInformer* event system along with the combinators which piece together the appropriate event pipes based simply on the logic of the combinator and the signature of the provided runnables. Furthermore, the runnable that is generated has been verified for semantic soundness. A new parent signature can be automatically synthesized and the parent is ready for further composition with its clean and well defined signature interface. Consider again the example in Figure 9.4. With only a few lines of code we have run three models in parallel automatically generating solution passing, upper bound passing and lower bound passing and appropriately accounting for the fact that one is a relaxation of the other two (see Figure 9.9). Furthermore, all of this is just works as it should without further thought from the user.

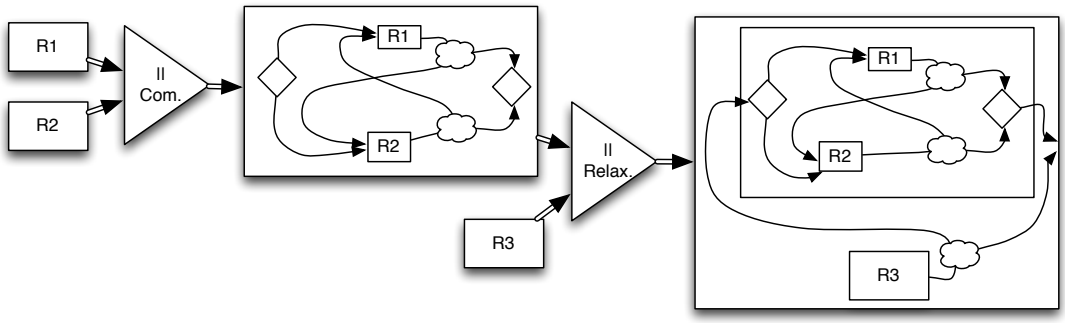

Figure 9.9: Parallel composition of two equivalent models with a relaxation. Multiple channels of communication generated automatically based on semantics contained within the relaxation relationships and signatures.

9.3 Transformations

Model transformations play a central part in Objective-CP and are crucial to the process of automating hybrids. Two of the most important kinds of transformations are:

Flatten : A transform which takes a high level model description and refactors it into a simplified form. This often involves refactoring algebraic expressions, introducing *views* and replacing modeling-level constraints with lower level constraints for which propagation algorithms exist. A *flattening* transformation is always applied to a model before concretization, but the particular transform depends on what concretization (CP, MIP, LS) will ultimately be instantiated. Flattening transforms are also sometimes applied before other transforms. For example, *reification constraints* are flattened prior to being linearized. Note that although this thesis makes substantial use of flattening operators, these operators were pre-existing.

Linearize : A linearization transform takes a high level model and transforms it into a linear equivalent suitable for use in a linear solver (MIP, IP, LP). This process involves introducing a *linear encoding* (section 1.4.2) for the model. For some constraints, a single 'most efficient' encoding exists and may be used. In other cases, such as in *scheduling*, different encodings may be desirable and specialized linearize operators exist (or may be implemented) for these cases. The development of linearization operators was an important part of the work that went into this thesis.

```
1 @protocol ORModelTransformation <NSObject>
2 -(void)apply:(id<ORModel>)m with:(id<ORAnnotation>)notes;
3 @end
```

Figure 9.10: Simple protocol implemented by model transformations

```
1 -(id<ORIntVarArray>) binarizeIntVar:(id<ORIntVar>)x
2 {
3     id<ORIntVarArray> o = [ORFactory intVarArray:_model range:[x domain] with:^id<ORIntVar>(
          ORInt i) {
4         return [ORFactory intVar:_model domain:RANGE(_model,0,1)];
5     }];
6     id<ORExpr> sumBinVars = Sum(_model, i,[x domain], o[i]);
7     id<ORExpr> sumExpr    = Sum(_model, i,[x domain], [o[i] mul: @(i)]);
8     [_model addConstraint: [sumBinVars eq: @(1)]];
9     [_model addConstraint: [sumExpr eq: x]];
10    return o;
11 }
```

Figure 9.11: Method to binarize integer variables as part of a linear transformation

Model Transformations all implement a simple protocol (Figure 9.10) and leverage the well-known *visitor pattern* on model *variables*, *constraints*, *objective* and *modeling objects*. Optionally, an *annotation* may be added to direct the transformation. This is used during CP concretization, for example, to select between different *propagation* algorithms.

To get an idea of how this process works, several linearization operators will be formally defined along with several Objective-CP implementation examples. Below is a spec for the *binarization* operator which is required for the linearization of many constraints:

$$B(x) = \langle B_C(x), B_X(x) \rangle, \text{ where}$$

$$B_X(x) \rightarrow \{b_{x=v_0} \dots b_{x=v_n}\}, n = |D(x)| \text{ and}$$

$$B_C(x) \rightarrow \left\{ \begin{array}{l} \sum_{v \in D(x)} b_{x=v} = 1, \\ \sum_{v \in D(x)} v \cdot b_{x=v} = x \end{array} \right\}$$

The operator binarizes an integer variable, x into a set of Boolean variables (one variable for each value in $D(x)$). It is assumed here that the array is also indexed by values in $D(x)$. Two constraints are present in $B_C(x)$, the first one requires that exactly one variable in the binarization be 1 (since the variable can only be assigned to a single value) and the second constraints links the value of x to the value of its binarization. Once a binarization has been created for a variable, it is

```
1 -(void) visitReifyNEqualc: (id<ORReifyNEqualc>)cstr
2 {
3     id<ORIntVar> x = [cstr x];
4     id<ORIntVar> r = [cstr b];
5     ORInt c = [cstr cst];
6
7     id<ORIntVarArray> bx = [self binarizeIntVar: x];
8     [_model addConstraint: [r eq: [@(1) sub: bx[c]]]];
9 }
```

Figure 9.12: linear transformation of a reification constraint $r \iff x \neq c$

cached by the transformation operator for reuse in future constraints. This is important as having numerous binarizations of the same variable can quickly lead to an unneccesary ballooning in the size of the model. Figure 9.11 shows this spec implemented in Objective-CP code.

The *constraint linearization transformation* may also be expressed formally with a collection of operators $L = \langle L_C, L_X \rangle$. Applying L to a constraint c (potentially non-linear), we get $L(c) = \langle L_C(c), L_X(c) \rangle$ where $L_C(c)$ is a mapping to a set of linear constraints and $L_X(c)$ is a mapping to a set of auxiliary variables required for the linearization.

Example (reification) Consider the linearization of the constraint *reify*: $r \iff x \neq c$. Formally, this linearization may be expressed as:

$$L^{\text{reify}}(r \iff x \neq c) = \langle L_C^{\text{reify}}(r \iff x \neq c), L_X^{\text{reify}}(r \iff x \neq c) \rangle, \text{ where}$$
$$L_X^{\text{reify}}(r \iff x \neq c) \rightarrow B(x) = \{b_{x=v_0} \dots b_{x=v_n}\}, \text{ and}$$
$$L_C^{\text{reify}}(r \iff x \neq c) \rightarrow \{r = 1 - b_{x=c}\} \cup B_C(x)$$

Here, $D(x) = \{v_0 \dots v_n\}$ and L_X^{reify} captures the Boolean variables introduced by the *binarization* of x. The first constraint in L_C^{reify} expresses the reification while $B_C(x)$ represents the constraints introduced by the binarization of x. Figure 9.12 shows this method implemented in Objective-CP. Lines 3-5 retrieve the variables and constant associated with the reify constraint. Line 7 gets the *binarization* of x and line 8 sets the result variable $r = 1 - b_{x=c}$ (where $b_{x=c}$ is the binarization of x at c).

Example (disjunctive scheduling) Objective-CP provides rich task variables for modeling scheduling problems which capture properties of a task such as *start time* and *duration* (as variables

within the task). Scheduling also has its own collection of *global constraints*, including:

precedes(t_1, t_2) Requires task t_1 to finish before task t_2 begins.

finished_by(t, x) Requires task t to finish before time x.

disjunctive(t_1, t_2) Requires that tasks t_1 and t_2 don't overlap.

The formal descriptions of these linearization methods are as follows:

$$L^{\text{precedes}}(precedes(t_1, t_2)) \rightarrow \langle\{\text{start}(t_1) + \text{duration}(t_1) \leq \text{start}(t_2)\}, \emptyset\rangle$$

$$L^{\text{finished\_by}}(finished\_by(t, x)) \rightarrow \langle\{\text{start}(t_1) + \text{duration}(t_1) \leq x\}, \emptyset\rangle$$

$$L^{\text{disjunctive}}(disjunctive(t_1, t_2)) \rightarrow \langle L_C^{\text{disjunctive}}(disjunctive(t_1, t_2)), L_X^{\text{disjunctive}}(disjunctive(t_1, t_2))\rangle,$$

$$\text{where } L_X^{\text{disjunctive}}(disjunctive(t_1, t_2)) \rightarrow \{z\} \text{ and}$$

$$L_C^{\text{disjunctive}}(disjunctive(t_1, t_2)) \rightarrow \left\{ \begin{array}{l} \text{start}(t_1) + \text{duration}(t_1) \leq \text{start}(t_2) + z \cdot M, \\ \text{start}(t_2) + \text{duration}(t_2) \leq \text{start}(t_1) + (1 - z) \cdot M \end{array} \right\}$$

The linearization of *precedes* leverages the existing start time and duration of the tasks, t_1 and t_2. The constraint $\text{start}(t_1) + \text{duration}(t_1) \leq \text{start}(t_2)$ requires that the start time plus the duration of t_1 is less than the start time of t_2. The linearization of *finished_by* is similar to that of *precedes*. In the *disjunctive* case, a 'big-M' formulation is employed, introducing two constraints requiring that either t_0 precedes t_1 or t_1 precedes t_0.

Figure 9.13 provides an OBJECTIVE-CP implementation of the transformation methods. The global constraint *precedes* is linearized on lines 13-20 and *finished_by* is linearized on lines 22-27. The *disjunctive* constraint is generalized here, accepting an array of task variables and requiring that no two overlap. Lines 1-11 provide a helper method producing the disjunction linearization for just two tasks. This helper method is used for linearizing the entire task array on lines 29-39.

Example (product) The final example will be the linearization of a product constraint: $r = b \cdot x$, where x and r are integer variables and b is a Boolean variable. Formally, this linearization may

```
1 -(void) noOverlap: (id<ORTaskVar>) t0 with: (id<ORTaskVar>) t1 {
2    id<ORIntVar> s0 = t0.getStartVar;
3    id<ORIntVar> s1 = t1.getStartVar;
4    ORInt d0 = t0.duration.up;
5    ORInt d1 = t1.duration.up;
6
7    ORInt M = /* large constant */;
8    id<ORIntVar> z = [ORFactory intVar: _model domain: RANGE(_model, 0, 1)];
9    [_model addConstraint: [[s0 plus: @(d0)] leq: [s1 plus: [z mul: @(M)]]]];
10   [_model addConstraint: [[s1 plus: @(d1)] leq: [s0 plus: [[@(1) sub: z] mul: @(M)]]]];
11 }
12
13 -(void) visitTaskPrecedes: (id<ORPrecedes>) cstr
14 {
15   id<ORTaskPrecedes> precedesCstr = (id<ORTaskPrecedes>)cstr;
16   id<ORIntVar> s0 = [[precedesCstr before] getStartVar];
17   ORInt d0 = [[[precedesCstr before] duration] up];
18   id<ORIntVar> s1 = [[precedesCstr after] getStartVar];
19   [_model addConstraint: [[s0 plus: @(d0)] leq: s1]];
20 }
21
22 -(void) visitTaskIsFinishedBy:  (id<ORTaskIsFinishedBy> ) cstr
23 {
24   id<ORIntVar> s0 = [[cstr task] getStartVar];
25   ORInt duration = [[[cstr task] duration] up];
26   [_model addConstraint: [[s0 plus: @(duration)] leq: [cstr date]]];
27 }
28
29 -(void) visitTaskDisjunctive: (id<ORTaskDisjunctive>) cstr
30 {
31   id<ORTaskVarArray> tasks = [cstr taskVars];
32   for(ORInt i = [tasks low]; i < [tasks up]; i++) {
33      for(ORInt j = i+1; j <= [tasks up]; j++) {
34         id<ORTaskVar> t0 = [tasks objectAtIndexedSubscript: i];
35         id<ORTaskVar> t1 = [tasks objectAtIndexedSubscript: j];
36         [self noOverlap: t0 with: t1];
37      }
38   }
39 }
```

Figure 9.13: linear transformation for scheduling

be expressed as:

$$L_X^{\text{prod}}(r = b \cdot x) \to \emptyset, \text{ and}$$

$$L_C^{\text{prod}}(r = b \cdot x) \to \left\{ \begin{array}{l} L \cdot b \leq r \leq U \cdot b, \\ r \leq x - L \cdot (1 - b), \\ r \geq x - U \cdot (1 - b), \\ D(x) = \{L \ldots U\} \end{array} \right\}$$

This linearization does not require any auxiliary variables, as L_X^{prod} produces an empty set. The transform produces three linear constraints, however, with constants L and U being the upper and

lower bounds of x respectively. The first constraint forces $r = 0$ whenever $b = 0$ and the last two constraints require $r = x$ whenever $b = 1$.

9.3.1 Linearizing Algebraic Constraints

The linearization of algebraic constraints requires a bit more work than the examples shown thus far. In particular, a linearizer for algebraic constraints needs to traverse the expression tree replacing non-linear subexpressions with linear equivalents. Consider the following constraint:

$$x_1 \leq (x_2 \neq 3) \cdot x_3 + 1$$

where x_1, x_2 and x_3 are integer variables. The inequality is clearly non-linear as it contains the product of a variable with a reification. Linearizing such an expression requires defining a new collection of operators, $\Theta = \langle \Theta_E, \Theta_C, \Theta_X \rangle$. Applying Θ to an expression e, yields the tuple $\langle \Theta_E(e), \Theta_C(e), \Theta_X(e) \rangle$, where:

$\Theta_E(e)$ is a linear expression equivalent to e.

$\Theta_C(e)$ is a set of new linear constraints introduced while linearizing e.

$\Theta_X(e)$ is the set of auxiliary variables created while linearizing e.

Each of these operators can be defined inductively for different types of expressions. For example:

$$\Theta(e_1 + e_2) \rightarrow \langle \Theta_E(e_1) + \Theta_E(e_2), \quad \Theta_C(e_1) \cup \Theta_C(e_1), \quad \Theta_X(e_1) \cup \Theta_X(e_1) \rangle$$
$$\Theta(e_1 - e_2) \rightarrow \langle \Theta_E(e_1) - \Theta_E(e_2), \quad \Theta_C(e_1) \cup \Theta_C(e_1), \quad \Theta_X(e_1) \cup \Theta_X(e_1) \rangle$$

defines linearization for addition and subtraction expressions. Analogous linearization rules can be defined for other binary relations. Similarly, for an expression multiplied by a constant, k, we may define:

$$\Theta(k \cdot e_1) \rightarrow \langle k \cdot \Theta_E(e_1), \quad \Theta_C(e_1), \quad \Theta_X(e_1) \rangle$$

Linear terminal nodes, such as a constant k or variable x, may be defined as:

$$\Theta(k) \to \langle k, \emptyset, \emptyset \rangle$$

$$\Theta(x) \to \langle x, \emptyset, \emptyset \rangle$$

The situation gets more interesting for non-linear terminal nodes. Consider the definition for the reification expression $x \neq k$:

$$\Theta(x \neq k) \to \langle \alpha, \quad L_C^{\text{reify}}(\alpha \iff x \neq k), \quad \{\alpha\} \cup L_X^{\text{reify}}(\alpha \iff x \neq k) \rangle$$
$$\text{where } D(\alpha) = \{0, 1\}$$

Here, α is a fresh Boolean variable forming a reification constraint with the expression $(x \neq k)$. The operators L_C^{reify} and L_X^{reify} are the linearization operators from the previous section producing the constraints and auxiliary variables required.

Similarly, we may define Θ for the product of an integer variable and Boolean variable:

$$\Theta(b \cdot x) \to \langle \alpha, \quad L_C^{\text{prod}}(b \cdot x = \alpha), \quad \{\alpha\} \cup L_X^{\text{prod}}(b \cdot x = \alpha) \rangle$$
$$\text{where } D(\alpha) = D(x) \cup \{0\}$$

Now, consider again the non-linear constraint, $x_1 \leq (x_2 \neq 3) \cdot x_3 + 1$. This expression may be linearized using Θ. The linear expression is computed as follows:

$$\Theta_E(x_1 \leq (x_2 \neq 3) \cdot x_3 + 1) \Rightarrow$$

$$\Theta_E(x_1) \leq \Theta_E((x_2 \neq 3) \cdot x_3 + 1) \Rightarrow$$

$$\Theta_E(x_1) \leq \Theta_E((x_2 \neq 3) \cdot x_3) + \Theta_E(1) \Rightarrow$$

$$\Theta_E(x_1) \leq \Theta_E(\Theta_E(x_2 \neq 3) \cdot x_3) + \Theta_E(1) \Rightarrow$$

$$x_1 \leq \Theta_E(\alpha_1 \cdot x_3) + 1 \Rightarrow$$

$$x_1 \leq \alpha_2 + 1$$

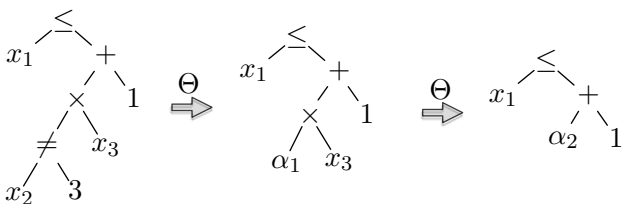

Figure 9.14: Linearization of expression tree for $x_1 \leq (x_2 \neq 3) \cdot x_3 + 1$

Figure 9.14 shows Θ applied to the expression tree for $x_1 \leq (x_2 \neq 3) \cdot x_3 + 1$. Note that Θ substitutes a Boolean variable (α_1) for the reification expression, yielding the product $\alpha_1 \cdot x_3$ as a subexpression. This subexpression is still not linear, but Θ is able to further reduce it, substituting the integer variable α_2 for the product. The end result brings together a number of the constraints and auxiliary variables we have seen already:

$$\Theta_E(x_1 \leq (x_2 \neq 3) \cdot x_3 + 1) \rightarrow x_1 \leq \alpha_2 + 1$$

$$\Theta_X(x_1 \leq (x_2 \neq 3) \cdot x_3 + 1) \rightarrow \{\alpha_1, \alpha_2, b_{x=v_0} \ldots b_{x=v_n}\}$$

$$\Theta_C(x_1 \leq (x_2 \neq 3) \cdot x_3 + 1) \rightarrow \left\{ \begin{array}{l} \alpha_1 = 1 - b_{x=c}, \\ \sum_{v \in D(x)} b_{x=v} = 1, \\ \sum_{v \in D(x)} v \cdot b_{x=v} = x \\ L \cdot \alpha_1 \leq \alpha_2 \leq U \cdot \alpha_1, \\ \alpha_2 \leq x - L \cdot (1 - \alpha_1), \\ \alpha_2 \geq x - U \cdot (1 - \alpha_1) \\ D(x) = \{L \ldots U\} \end{array} \right\}$$

In particular, Θ_X maps our constraint to a set of auxiliary variables containing the binarization for x (needed for the reification) as well as α_1 and α_2 introduced by Θ as substitutions for the product and reification subexpressions. The set of linear constraints produced by Θ_C contains the two constraints from the binarization of x as well as the three constraints produced by the linearization of the product. Note that the constraints and variables from $B(x)$ are explicitly stated here for completeness.

The operator Θ may be expanded to include rules for linearizing any expression. The current implementation of this operator in OBJECTIVE-CP traverses an expression tree using the *visitor pattern* and includes many rules not shown here. Using Θ, we may also define a linearization operator for algebraic constraints as follows:

$$L^{\text{algebra}}(e) = \langle L_C^{\text{algebra}}(e), L_X^{\text{algebra}}(e) \rangle, \text{ where}$$
$$L_X^{\text{algebra}}(e) \rightarrow \Theta_X(e),$$
$$L_C^{\text{algebra}}(e) \rightarrow \Theta_C(e) \cup \{\Theta_E(e)\}$$

9.4 Transcoding Solutions

Solution transcoding is a difficult problem in its own right, but fortunately, OBJECTIVE-CP is built from the ground up on sophisticated model transformation which preserve mapping information (referred to as γ) on related models (for a detailed overview of this refer to a reference on OBJECTIVE-CP such as [94]). In particular, consider an abstract model M concretized into two programs P_{CP} and P_{MIP}. Recall that the user is only responsible for creating the model M, while P_{CP} and P_{MIP} are generated by opaque concretization operators. Hence, OBJECTIVE-CP is designed to implicitly map a solution σ_{CP} (from P_{CP}) 'up' to a solution σ for M (Figure 9.15, left). The need to map solutions 'down' (Figure 9.15, right), however, only became necessary when engineering solution communication between solvers for hybrid *runnables*.

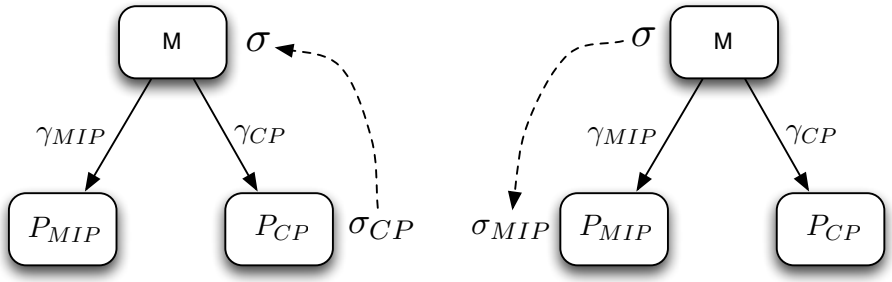

Figure 9.15: Transcoding a solution 'up' from a low level representation to a high level one (left) and from a high level representation 'down' to a low level one (right).

Transcoding of solutions between solvers is handled transparently within the *runnable* produced by the *parallel model combinator* allowing two runnables R_0 and R_1 running in two distinct threads to cooperate transparently. Recall that transcoding 'up' is implicit in OBJECTIVE-CP. The fragment below describes how to transcode 'down'.

```
1 transcode(Runnable t,Solution σ) → σ′ {
2     σ′ := ∅
3     forall x in vars(σ):
4         {⟨y_i ↦ v_i⟩} := decode(x,σ(x),t)
5         σ′ := σ′ ∪ {⟨y_i ↦ v_i⟩}
6     t.inject(σ′)
7     return σ′
8 }
```

Given a solution σ and a target runnable t, it creates a new concrete solution σ' adapted to t's encoding. The loop on line 3 iterates over all the variables in σ and *decodes* with the call on line 4 the assignment to variable x in term of its representation in t. If t relies on an encoding of x with domain $D(x) = \{0 \cdots n\}$ into $n+1$ binary variables $x_0 \cdots x_n$, the decoder produces a collection of assignments to cover all the binary variables y_i of t (only one of which is assigned 1). Line 6 installs the solution σ' in t and returns it.

The ability to stream solutions from R_0 to R_1 is critical. Gurobi, for instance, cannot tighten its upper bound to a new incumbent bound f^* and instead mandates the communication of the entire solution which must be installed and validated to get the bound. A Gurobi runnable must therefore consume solutions from a callback invoked at each node of its search tree.

9.5 Parameterized Models & Lagrangian Operators

Parameterized models and lagrangian operators (see Chapter 8) enable generic LR runnables. Figure 9.16 shows the relevant protocols. *ORParameter* (lines 11-14) captures the concept of a parameter at the modeling level. *ORParameters*, like *ORVars*, are expressions in OBJECTIVE-CP and may be used within any algebraic constraint or as part of a larger expression. *ORParameters* do not take on any value within the model itself, but instead are set by the user for each concrete program derived from the model. Within a program, parameters behave exactly like constants, but

```
1 @protocol ORParameterizedModel <ORModel>
2 -(NSArray*) softConstraints;
3 -(NSArray*) hardConstraints;
4 -(NSArray*) parameters;
5 -(id<ORVarArray>) slacks;
6 -(void) addParameter: (id<ORParameter>)p;
7 -(id<ORWeightedVar>) parameterization: (id<ORVar>)slackVar;
8 -(id<ORWeightedVar>) parameterizeVar: (id<ORVar>)slackVar;
9 @end
10
11 @protocol ORParameter <ORObject,ORExpr>
12 -(ORUInt) getId;
13 @end
14
15 @protocol ORSoftConstraint <ORConstraint>
16 -(id<ORVar>)slack;
17 @end
18
19 @protocol ORWeightedVar <ORConstraint>
20 -(id<ORVar>) z;
21 -(id<ORVar>)x;
22 -(id<ORParameter>)weight;
23 @end
```

Figure 9.16: Protocol for parameterized models in Objective-CP.

may be modified between runs of the program.

The *ORParameterizedModel* (lines 1-9) is an extension of the standard *ORModel*, capturing several new concepts. Lines 2-3 show that constraints are split into 'hard' and 'soft' variations. Hard constraints are constraints that will trigger a failure if not satisfied. Soft constraints (lines 15-17) are extensions of a standard constraint, but rather than triggering failure, they provide a measure of satisfaction or violation in the form of a *slack* variable (line 16). Lines 4, provided all model parameters and line 5 provides all slack variables associated with soft constraints in the model.

The concept of an *ORWeightedVar* (lines 19-23), allows the slack variable of a soft constraint to take the form of a weighted penalty in the objective function. *ORWeightedVar* is, in fact, a constraint of the form $z = weight \cdot x$, where z is a penalty variable appended to the objective function, *weight* is a cost parameter and x is a slack variable associated with some soft constraint. *ORParameterizedModel* provides methods for generating an *ORWeightedVar* (given a slack, line 8) and for looking up an *ORWeightedVar* (line 7).

The architecture, presented in figure 9.16 enables LR to be captured in a generic way. In particular, the notion of a soft constraint is general enough to apply to a 'relaxed inequality' of the kind typically used in LR as well as softened constraints from CP or LS.

```
1 id<ORModel> m = /* problem model */
2 NSArray* relaxedCstrs = /* set of constraints to relax */
3 id<ORParameterizedModel> lrm = [[ORFactory lagrangianSLRTransform] apply: m relaxing:
      relaxedCstrs];
4 id<ORRunnable> r = [ORFactory CPSubgradient: lrm bound: UB];
```

Figure 9.17: Producing a parameterized model for LR from a standard model.

Producing a *ORParameterizedModel* for use with LR from a standard *ORModel* is straightforward in Objective-CP. Figure 9.17 demonstrates the process in several lines of code. Line 1 creates a high-level description of the problem as an *ORModel* and line 2 defines a set of constraints the user would like to relax. Line 3 transform the original model and the set of relaxed constraints into a new *ORParameterizedModel* represent an SLR formulation (see Chapter 8) of the problem. The last line produces a CP runnable by plugging the parameterized SLR model into a subgradient template with an upper bound. The implementation of the SLR and GLR transforms is built on the *visitor pattern* with the same approach as the *linearization* operators in section 9.3.

Figure 9.18 shows how *ORParameterizedModel* is used to implement the subgradient method (see section 8.2). Lines 8-12 collect the lambdas into an array for easy updating during the iterative procedure. In particular, line 10 uses the parameterized model to look up the *ORWeightedVar* for a slack variable and line 11 returns the associated weight parameter. Lines 15-53 contain the subgradient iteration loop. The program is solved on line 17 and an optimal solution and bound are captured on lines 19 and 20. Lines 22-28 sum up the values of the slack variables in the solution and check if any are positive (indicating the original problem is not satisfied). Lines 30-39 update subgradient parameters and the best solution and bound if needed. Lines 42-49 update the lambdas in preparation for the next iteration of the method. In detail, lines 44-47 compute the new lambda value and line 48 sets the new value for the lambda in the program.

```
1 -(id<ORSolution>) subgradientSolve(id<ORParameterizedModel> model, id<ORASolver> program) {
2     ORDouble slackSum = 0.0;
3     id<ORParameterizedSolution> bestSol = nil;
4     id<ORVarArray> slacks = [model slacks];
5     id<ORIntRange> slackRange = [slacks range];
6
7     // Collect lambdas in array
8     id<ORIdArray> lambdas = [ORFactory idArray: model range: slackRange with: ^id(ORInt i) {
9         id<ORVar> slack = [slacks at: i];
10        id<ORWeightedVar> w = [model parameterization: slack];
11        return [w weight];
12    }];
13
14    // Start subgradient iteration loop
15    while(pi > cutoff) {
16        [[program solutionPool] emptyPool];
17        [self solveIt: program];
18
19        id<ORParameterizedSolution> sol = [[program solutionPool] best];
20        ORDouble bound = [[sol objectiveValue] doubleValue];
21
22        BOOL satisfied = YES;
23        slackSum = 0.0;
24        [slacks enumerateWith:^(id<ORVar> var, ORInt idx) {
25            ORDouble s = [sol doubleValue: var];
26            slackSum += s * s;
27            if(s > 0) satisfied = NO;
28        }];
29
30        // Check for improvement
31        if(bound > _bestBound) {
32            _bestBound = bound;
33            bestSol = sol;
34            noImprove = 0;
35        }
36        else if(++noImprove > noImproveLimit) {
37                pi /= 2.0;
38                noImprove = 0;
39        }
40
41        // Update lambdas
42        ORDouble stepSize = pi * (_ub - bound) / slackSum;
43        [lambdas enumerateWith:^(id<ORRealParam> lambda, ORInt idx) {
44            ORDouble value = [sol paramValue: lambda];
45            id<ORVar> slack = [slacks at: idx];
46            ORDouble slackVal = [sol doubleValue: slack];
47            ORDouble newValue = MAX(0, value + stepSize * slackVal);
48            [program param: lambda setValue: newValue];
49        }];
50
51        // Check if done
52        if(satisfied) break;
53    }
54    return bestSol;
55 }
```

Figure 9.18: Subgradient method implemented with *ORParameterizedModel*.

Chapter 10

Conclusion

10.1 Thesis Review

Discrete Optimization is a broad field dealing with problems that are increasingly relevant in the modern world, yet exceedingly difficult to solve computationally. Several generic technologies have been developed over the years to deal with these problems, including Constraint Programming (CP), Mixed-Integer Programming (MIP), SAT and local search methods. These technologies are declarative in nature, allowing users to model problems and then apply various solution methodologies. Beyond these standalone technologies, a broad literature of hybrid techniques has developed. Hybrids employ multiple optimization techniques within a single solver, sometime also integrating multiple optimization technologies. As the scale of combinatorial problems have increased, hybrid techniques have become increasingly prominent both in research and in practice. Yet, relatively little research has been done on defining the abstractions and semantics necessary to make these techniques readily available for users in a plug-and-play and composable fashion. This section will review the contributions this thesis has made towards this effort.

1. The first contribution of this thesis was to define a vision and initial implementation for a hybrid automation platform. Compare the literature at the time (Chapter 4) with the proposal and implementation of *CML* (chapters 2 and 5). *CML* articulated a vision of an algebra of operators over abstract models in which operators could be chained together to

build up arbitrarily complex hybrids. This included operators for various purposes, including:

- Transforming abstract models between various problem formulations.
- Concretizing abstract models into technology specific solvable models.
- Applying searches to models (including LNS and custom searches).
- Combining models into hybrids.

Additionally, *CML* provided an initial implementation of this platform as a custom scripting language on top of COMET allowing complex hybrids to be constructed with only a couple lines of code. Examples included a bounds sharing parallel hybrid combining MIP with CBLS (a first at the time), a sequential hybrid combing a CBLS solver with a CP solver derived from the same high level model, a template-based column generation hybrid and various LNS-based solvers.

The implementation of *CML* served as a prototype demonstrating that the complex communication, synchronization and solution transcoding that underlies many hybrid techniques could be effectively automated with little or no overhead.

2. Next, the ideas underlying *CML* were directly implemented within the OBJECTIVE-CP optimization library (Chapter 5) and greatly extended as *model combinators*. New concepts included:

   - The definition of relationships between models, tightenings and relaxations, which are derived through model transformations. The transitive closures of these relationships enables us to verify preconditions on the models. Combinators can also specify their relationship with the underlying models.
   - The concept of runnable and runnable signatures that specify the functionalities supported by an optimization program and its model.
   - The concept of model specifications, including input/output/internal pipes and pipe rules, that enables the synthesis of the signature of the combinators from the signature of their components.

Model combinators removed the semantic ambiguity inherent in *CML* allowing models to be truly composible and provided a systematic mechanisms for synthesizing input/output interfaces and verifying preconditions on models. A number of model combinators were introduced, including sequential and parallel composition, column generation, and Benders decomposition. Case studies on Logic-Based Benders decomposition indicated that a hand written solver for the warehouse location-allocation problem could be efficiently automated.

3. Lagrangian relaxation, a classic technique in continuous optimization, was generalized to arbitrary high-level models (Chapter 8). Lagrangian relaxations of such models can be concretized into a variety of optimization technologies (constraint programming, local search, or MIP). The resulting concrete optimization programs can be entrusted to algorithmic templates to solve Lagrangian duals or use Lagrangian primal methods. This thesis also demonstrated that lagrangian relaxations can be built around the notion of satisfiability degrees (typical in mathematical programming) or violation degrees (typically in constraint programming and local search). Finally, this work indicated how to apply surrogate optimization systematically in a generic algorithmic template that optimizes independent problems separately. Case studies showed the versatility of lagrangian relaxation for a variety of solver technologies and models. In particular, they show that

    - The Lagrangian dual coupled with constraint programming is an effective method for some classes of graph coloring problems.
    - The concept of violation degree is valuable to improve the quality and performance of the Lagrangian dual when solved with MIP solvers. It is not clear however whether satisfiability degrees can be valuable for constraint programming or local search.
    - Primal Lagrangian methods may systematically boost the performance and solution quality of meta-heuristics in a principled way.

   Overall, these results indicated that Lagrangian methods could play a much more significant role in constraint programming and large neighborhood search.

4. *Cooperative portfolio solvers* were introduced by extending *model combinators* to scheduling

(Chapter 6). Case studies demonstrated that cooperative solvers including CP/MIP, CP/LNS and MIP/LNS could substantially outperform standalone CP and MIP solvers on numerous standard jobshop benchmarks. An analysis on the effects of parallelism (single thread, two threads, 4 threads) within standalone solvers demonstrated an unsettling lack of robustness, with more parallelism often leading to worse overall performance. Portfolio solvers were shown to be an effective means of achieving robustness in a parallel setting.

5. A new LNS heuristic for the IMRT problem was introduced (Chapter 7). The heuristic allows the *Counter Model* to find high quality solutions on much larger instance sizes than is possible with a complete search. This approach typically found solutions within 10% of the optimal several times faster than a complete search and found the optimum in almost half of the instances. The LNS approach relied on profiling of a "typical solution" as training data allowing variables to be fixed to values likely to give quality solutions. The robustness of this approach was demonstrated on two qualitatively different classes of random instances. The heuristic was demonstrated to be easily implemented using the LNS operator provided by *CML*.

10.2 Future Work

The framework proposed and implemented in this thesis is general enough to encompass a broad array of hybrid techniques. Hence, there will always be the possibility of digging up a new technique from the literature and generalizing it into a *model combinator*. There is also the possibility of integrating additional technologies such as SAT and SMT into the framework which would allow for new parallel and sequential compositions. Another promising path is to investigate how the communication of *nogoods* could be integrated into the *parallel combinator*.

One topic that particularly interests me arose while investigating *cooperative portfolio solvers*. As discussed in chapter 6, standard search space splitting techniques employed by CP to exploit parallelism often lack robustness. Ideally, the technique should produce a linear speedup, proportional to the number of CPU cores. Unfortunately, this is often not the case, with performance being highly dependent on the problem instance and search strategy employed. It is common to

encounter instances where the benefits of parallelism are negligible or even cases with a significant degradation in performance.

Cooperative portfolio solvers suggested a method for regaining robustness. That is, rather than splitting the search space and changing the node ordering in unpredictable ways, each core can be given its own independent solver from the portfolio and communication can be set up seamlessly. Such an approach even holds out the possibility of super-linear speedup (as suggested recently in [31]) as the solvers can share bounds, helping to rapidly prune each others search spaces in unpredictable ways. The difficulty with this approach is that it only effective if you have different, high quality models to distribute to every core. In Chapter 6, we were fortunate to have three very different, yet competitive models for jobshop in the literature that had never been combined in parallel before. Generally, however, it will not be the case that there is a pre-existing portfolio of quality models for a given problem, particularly as we look towards a future where 8 and 16 cores become commonplace.

One path is to turn towards the machine learning techniques employed by standard portfolio solver to generate a set of solvers (see [8], [46]). Another approach is to take one good problem formulation and to systematically relax parts of the problems to provide models for other cores. The relaxations could be combined in parallel with each other (where appropriate) and also with a solver running on the full problem. The relaxations would provide lower bound to the full problem (potentially very useful in CP). Additionally, optimal solutions on a relaxation could be used as the basis of variable and value ordering heuristic on both the full problem and less relaxed solvers. Initially, a set of very aggressively relaxed problems would be run allowing optimal solutions to be found quickly. As the relaxations are solved, they could be incrementally tightened and the search on the tightened problem could be seeded with high quality solutions from the predecessor.

This leaves open the question of how to systematically relax a problem. There is a broad literature on relaxation techniques, some of them specific to particular problems. A relaxation technique is needed that applies to any problem and also allows a relatively fined grained levels of relaxations. This is desirable so we can be assured that we can create enough relaxations to keep worker threads busy. Also, by creating closely related relaxations, the optimal solution to one relaxation is more likely to be related to the optimal solution of its successor making a search

strategy based on the previous optimum more effective.

The substance of this project would clearly be in the development of such a relaxation and the related variable/value ordering heuristic as the *parallel combinator* make implementing such a complex solver trivial. One potential approach is what could be called a *no-fail relaxation* in CP. Most relaxation techniques are based on removing or otherwise softening difficult constraints within a problem. A *no-fail relaxation* would take the approach of identifying a set of hard variables, X_H, which would be prevented from triggering a failure in a CP search. Typically a failure occurs at a node when some variable domain becomes empty during propagation. A variable $x \in X_H$, could be prevented from triggering a failure and instead could be frozen to its last non-empty domain, $F(x)$, at the point it would have otherwise become empty. The search would be allowed to proceed, but the frozen domain would no longer participate in pruning related events. The domain would remain frozen unless the search backtracked to a point before it became frozen. Hence, the search would likely finish with variables in X_H unbound (frozen domains of size > 1). Since the relaxation must produce an objective value less than or equal to the optimal of the original problem, each variable $x \in X_H$ would take on the value $\min F(x)$ in the objective function.

Obviously, until the relaxation is implemented, it is unclear how effective it will be. There is reason to think the relaxation might become hard to solve as $|X_H| \to 0$ because the process of freezing domains eliminates some propagation. That is, because a frozen domain can't participate in pruning events, the CP arc consistency algorithm may terminate prematurely resulting in less overall pruning. Therefore, there may be a point at which the benefits of having a 'relaxed' problem are outweighed by the loss of propagation.

The overall goal, however, of systematically generating related relaxations to run in a massively parallel setting is promising and something I hope to explore in detail in the future.

Bibliography

[1] Y. Tourbier A. Oplobedu, J. Marcovitch. Charme: Un langage industriel de programmation par contraintes, illustré par une application chez renault. In *Proceedings of the Ninth International Workshop on Expert Systems and their Applications: General Conference 1*, pages 55–70, 1989.

[2] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[4] Ozgur Akgun. Extensible automated constraint modelling via refinement of abstract problem specifications. 2014.

[5] Ozgur Akgun, Ian Miguel, Chris Jefferson, Alan Frisch, and Brahim Hnich. Extensible automated constraint modelling, 2011.

[6] Maria Albareda-Sambola, Elena Fernández, and Gilbert Laporte. The capacity and distance constrained plant location problem. *Computers and Operations Research*, 36(2):597 – 611, 2009. Scheduling for Modern Manufacturing, Logistics, and Supply Chains.

[7] Dionne M. Aleman, Arvind Kumar, Ravindra K. Ahuja, H. Edwin Romeijn, and James F. Dempsey. Neighborhood search approaches to beam orientation optimization in intensity modulated radiation therapy treatment planning. *J. of Global Optimization*, 42:587–607, December 2008.

[8] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny-cp: A sequential cp portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1861–1867, New York, NY, USA, 2015. ACM.

[9] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[10] Davaatseren Baatar, Natashia Boland, Sebastian Brand, and Peter J. Stuckey. *Minimum Cardinality Matrix Decomposition into Consecutive-Ones Matrices: CP and IP Approaches*, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2005. Research Report SICS T2005-08.

[12] Nicolas Beldiceanu and Thierry Petit. Cost evaluation of soft global constraints. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 80–95. Springer Berlin Heidelberg, 2004.

[13] J. F Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(3):238–252, 1962.

[14] Dimitris Bertsimas and John Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1st edition, 1997.

[15] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 146–150, Amsterdam, The Netherlands, The Netherlands, 2004. IOS Press.

[16] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[17] Hadrien Cambazard, Eoin O'Mahony, and Barry O'Sullivan. *A Shortest Path-Based Approach to the Multileaf Collimator Sequencing Problem*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[18] C. W. Choi, W. Harvey, J. H. M. Lee, and P. J. Stuckey. *Finite Domain Bounds Consistency Revisited*, pages 49–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[19] Kenneth M. F. Choi, Jimmy Ho-Man Lee, and Peter J. Stuckey. A lagrangian reconstruction of genet. *Artif. Intell.*, 123(1-2):1–39, 2000.

[20] Kenneth M.F. Choi, Jimmy H.M. Lee, and Peter J. Stuckey. A lagrangian reconstruction of {GENET}. *Artificial Intelligence*, 123(1–2):1 – 39, 2000.

[21] André Ciré, Elvin Coban, and John N. Hooker. *Mixed Integer Programming vs. Logic-Based Benders Decomposition for Planning and Scheduling*, pages 325–331. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[22] C. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In *AAAI Fall Symposium on Using Uncertainty within Computation*, Cape Cod, MA., 2001.

[23] Philippe Codognet and Daniel Diaz. *Stochastic Algorithms: Foundations and Applications: International Symposium, SAGA 2001 Berlin, Germany, December 13–14, 2001 Proceedings*, chapter Yet Another Local Search Method for Constraint Solving, pages 73–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[24] M. Conforti, G. Cornuejols, and G. Zambelli. *Integer Programming*. Graduate Texts in Mathematics. Springer International Publishing, 2014.

[25] Laurent Michel Daniel Fontaine. A large-scale neighborhood search approach to matrix decomposition into consecutive-ones matrices. 8th Workshop on Local Search techniques in Constraint Satisfaction, 9 2011.

[26] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[27] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[28] Gregory Duck, Leslie De Koninck, and Peter Stuckey. Cadmium: An implementation of acd term rewriting. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 531–545. Springer Berlin / Heidelberg, 2008.

[29] Gregory Duck, Peter Stuckey, and Sebastian Brand. Acd term rewriting. In Sandro Etalle and Miroslaw Truszczynski, editors, *Logic Programming*, volume 4079 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin / Heidelberg, 2006.

[30] Thorsten Ehlers and Peter J. Stuckey. *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016,*

Proceedings, chapter Parallelizing Constraint Programming with Learning, pages 142–158. Springer International Publishing, Cham, 2016.

[31] Thorsten Ehlers and Peter J. Stuckey. *Parallelizing Constraint Programming with Learning*, pages 142–158. Springer International Publishing, Cham, 2016.

[32] MohammadM. Fazel-Zarandi and J.Christopher Beck. Solving a location-allocation problem with logic-based benders' decomposition. In IanP. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 344–351. Springer Berlin Heidelberg, 2009.

[33] Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. *Half Reification and Flattening*, pages 286–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[34] M. Fisher, K.O. Joernsten, and O.B.G Madsen. Vehicle routing with time windows: Two optimization algorithms. *Operations Research*, 45(3):488–492, 1997.

[35] Marshall L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Manage. Sci.*, 50(12 Supplement):1861–1871, December 2004.

[36] ML Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 1981.

[37] Daniel Fontaine, LaurentMichel, and Pascal Van Hentenryck. *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, chapter Constraint-Based Lagrangian Relaxation, pages 324–339. Springer International Publishing, Cham, 2014.

[38] Daniel Fontaine and Laurent Michel. A high level language for solver independent model manipulation and generation of hybrid solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, volume 7298 of *Lecture Notes in Computer Science*, pages 180–194. Springer Berlin Heidelberg, 2012.

[39] Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. Model combinators for hybrid optimization. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 299–314. Springer Berlin Heidelberg, 2013.

[40] Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings*, chapter Parallel Composition of Scheduling Solvers, pages 159–169. Springer International Publishing, Cham, 2016.

[41] Alan Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[42] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000, December 2008.

[43] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.

[44] J. N. Hooker. A search-infer-and-relax framework for integrating solution methods. In *Proceedings of the Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'05, pages 243–257, Berlin, Heidelberg, 2005. Springer-Verlag.

[45] J.N. Hooker. Logic-based benders decomposition. *Mathematical Programming*, 96:2003, 1995.

[46] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, volume 8451 of *Lecture Notes in Computer Science*, pages 301–317. Springer International Publishing, 2014.

[47] Jonathan P. Seldin J. Roger Hindley. *Lambda-Calculus and Combinators An Introduction*. Cambridge University Press, 2nd edition, 2008.

[48] Serdar Kadioglu, Eoin O'Mahony, Philippe Refalo, and Meinolf Sellmann. Incorporating variance in impact-based search. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, CP'11, pages 470–477, Berlin, Heidelberg, 2011. Springer-Verlag.

[49] Regina Klimmek and Frank Wagner. A simple hypergraph min cut algorithm, 1996.

[50] Wen-Yang Ku and J Christopher Beck. Revisiting off-the-shelf mixed integer programming and constraint programming models for job shop scheduling. Technical report, University of Toronto, https://www.mie.utoronto.ca/research/technical-reports/reports/JSP.pdf, 2014.

[51] D. G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, Reading, Massachusetts, 1973.

[52] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[53] L. Michel, A. See, and P. Van Hentenryck. Transparent Parallelization of Constraint Programming. *INFORMS Journal on Computing*, 21(3):363–382, 2009.

[54] Laurent Michel and Pascal Van Hentenryck. A decomposition-based implementation of search strategies. *ACM TRANSACTIONS ON COMPUTATIONAL LOGIC*, 5(2), 2004.

[55] Laurent Michel and Pascal Van Hentenryck. *The Comet Programming Language and System*, pages 881–881. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[56] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'12, pages 228–243, Berlin, Heidelberg, 2012. Springer-Verlag.

[57] Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper. Parallel discrepancy-based search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 30–46. Springer Berlin Heidelberg, 2013.

[58] A. Nareyek. Using global constraints for local search. In E. C. Freuder and R. J. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *American Mathematical Society Publications*, pages 9–28. DIMACS, 2001.

[59] Alexander Nareyek. Using global constraints for local search. In *DIMACS Workshop on on Constraint Programming and Large Scale Discrete Optimization*, pages 9–28, Boston, MA, USA, 2001. American Mathematical Society.

[60] MohammadMahdi Nasiri and Farhad Kianfar. A guided tabu search/path relinking algorithm for the job shop problem. *The International Journal of Advanced Manufacturing Technology*, 58(9-12):1105–1113, 2012.

[61] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York, NY, USA, 1988.

[62] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *19th Irish Conference on AI*, 2008.

[63] J. P. Carillon P. Van Hentenryck. Generality versus specificity: An experience with ai and or techniques. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 660–664, 1988.

[64] Dario Pacino and Pascal Van Hentenryck. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. In *IJCAI*, pages 1997–2002, 2011.

[65] L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*, pages 346–360, Alexandra, VA, October 1999. Springer-Verlag.

[66] Laurent Perron, Paul Shaw, and Vincent Furnon. *Propagation Guided Large Neighborhood Search*, pages 468–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[67] Gilles Pesant. *A Regular Language Membership Constraint for Finite Sequences of Variables*, pages 482–495. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[68] D. Pisinger and S. Ropke. *Handbook of Metaheuristics International Series in Operations Research & Management Science*, volume 146, chapter Large Neighborhood Search, pages 399–419. Springer, 2010.

[69] Jakob Puchinger, Peter J. Stuckey, Mark Wallace, and Sebastian Brand. From high-level model to branch-and-price solution in g12, 2008.

[70] Jakob Puchinger, PeterJ. Stuckey, MarkG. Wallace, and Sebastian Brand. Dantzig-wolfe decomposition and branch-and-price solving in g12. *Constraints*, 16(1):77–99, 2011.

[71] Claude-Guy Quimper, Alexander Golynski, Alejandro López-Ortiz, and Peter Van Beek. An efficient bounds consistency algorithm for the global cardinality constraint. *Constraints*, 10(2):115–135, 2005.

[72] P. Refalo. Linear Formulation of Constraint Programming Models and Hybrid Solvers. In *Sixth International Conference on the Principles and Practice of Constraint Programming (CP'00)*, pages 369–383, Singapore, September 2000.

[73] Philippe Refalo. Linear formulation of constraint programming models and hybrid solvers. In Rina Dechter, editor, *Principles and Practice of Constraint Programming CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 369–383. Springer Berlin / Heidelberg, 2000.

[74] Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

[75] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 596–610. Springer Berlin Heidelberg, 2013.

[76] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[77] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter Stuckey. Search combinators. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming â“ CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 774–788. Springer Berlin / Heidelberg, 2011.

[78] Christian Schulte. Parallel search made simple. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, Singapore, September 2000.

[79] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26 of *American Mathematical Society Publications*. DIMACS, 1996.

[80] Yi Shang and Benjamin Wah. A Discrete Lagrangian-Based Global-Search Method for Solving Satisfiability Problems. *Journal of Global Optimization*, 12:61–99, 1998.

[81] Yi Shang and Benjamin W. Wah. A discrete lagrangian-based global-searchmethod for solving satisfiability problems. *J. of Global Optimization*, 12(1):61–99, January 1998.

[82] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP'98*, pages 417–431, 1998.

[83] Paul Shaw. *A Constraint for Bin Packing*, pages 648–662. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[84] Peter Stuckey, Maria de la Banda, Michael Maher, Kim Marriott, John Slaney, Zoltan Somogyi, Mark Wallace, and Toby Walsh. The g12 project: Mapping solver independent models to efficient solutions.

In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming*, volume 3668 of *Lecture Notes in Computer Science*, pages 9–13. Springer Berlin / Heidelberg, 2005.

[85] T Sun, Q C Zhao, and P.B. Luh. On the Surrogate Gradient Algorithm for Lagrangian Relaxation. *Journal of Optimization Theory and Applications*, 133(3):413–416, June 2007.

[86] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur (Belgium), July 1987.

[87] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[88] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[89] P. Van Hentenryck and L. Michel. Synthesis of constraint-based local search algorithms from high-level models. *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, 1(CONF 22):273–279, 2007.

[90] Pascal Van Hentenryck. *The OPL Optimization Programming Language.* MIT Press, Cambridge, MA, USA, 1999.

[91] Pascal Van Hentenryck, Yves Deville, and Choh Teng. A generic arc consistency algorithm and its specializations. Technical report, Providence, RI, USA, 1991.

[92] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search.* The MIT Press, 2005.

[93] Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. In *Proceedings of the Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'05, pages 380–395, Berlin, Heidelberg, 2005. Springer-Verlag.

[94] Pascal Van Hentenryck and Laurent Michel. *The Objective-CP Optimization System*, pages 8–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[95] Pascal Van Hentenryck and Laurent Michel. The objective-cp optimization system. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 8–29. Springer Berlin Heidelberg, 2013.

[96] Pascal Van Hentenryck and Laurent Michel. *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, chapter The Objective-CP Optimization System, pages 8–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[97] Petr Vilím, Roman Barták, and Ondřej Čepek. Extension of o(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 10(4):403–425, October 2005.

[98] Petr Vilim, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In Laurent Michel, editor, *Integration of AI and OR Techniques in Constraint Programming*, volume 9075 of *Lecture Notes in Computer Science*, pages 437–453. Springer International Publishing, 2015.

[99] Benjamin W. Wah and Zhe Wu. The theory of discrete lagrange multipliers for nonlinear discrete optimization. In *5th International Conference on the Principles and Practice of Constraint Programming - CP'99, , Alexandria, Virginia, USA, October 11-14, 1999.*

[100] Zhe Wu and Benjamin W. Wah. Trap escaping strategies in discrete lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In Jim Hendler and Devika Subramanian, editors, *AAAI/IAAI*, pages 673–678. AAAI Press / The MIT Press.

[101] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[102] Tallys Yunes, Ionuţ D. Aron, and J. N. Hooker. An integrated solver for optimization problems. *Oper. Res.*, 58(2):342–356, March 2010.

[103] Xing Zhao, P.B. Luh, and Jihua Wang. The surrogate gradient algorithm for Lagrangian relaxation method. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, pages 305–310, 1997.