UPPSALA THESES IN COMPUTING SCIENCE
33

Integration of Constraint Programming and Integer Programming for Combinatorial Optimization

Greger Ottosson

Uppsala University
Information Technology
Computing Science Department

Thesis for the Degree of
Doctor of Philosophy

UPPSALA 2000

Integration of Constraint Programming and Integer Programming for Combinatorial Optimization

Greger Ottosson

A Dissertation submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy at Computing Science Department, Information Technology, Uppsala University.

Computing Science Department
Information Technology
Uppsala University
Box 311, SE–751 05 Uppsala, Sweden

Uppsala Theses in Computing Science 33
ISSN 0283–359X
ISBN 91–506–1396–0

(Dissertation for the Degree of Doctor of Philosophy in Computing Science presented at Uppsala University in 2000)

Abstract

Ottosson, G. Integration of Constraint Programming and Integer Programming for Combinatorial Optimization, *Uppsala Theses in Computing Science 33*. 143 pp. Uppsala. ISSN 0283–359X, ISBN 91–506–1396–0.

The last several years have seen an increasing interest in combining the models and methods of optimization with those of constraint programming. Integration of the two was initially impeded by their different cultural origins, one having developed largely in the operations research community and the other in the computer science and artificial intelligence communities. The advantages of merger, however, are rapidly overcoming this barrier.

The main objective for an integration of Constraint Programming over finite domains (CP) and Integer Programming (IP) is to take advantage of both the inference through constraint propagation and the (continuous) relaxations through Linear Programming (LP), in order to reduce the search needed to find feasible, good and optimal solutions.

The key decisions to be made for integrating CP and IP are (a) the model(s), (b) the inference, (c) the relaxations, and, (d) the search and branching strategies to use. In this thesis it is advocated to model specifically for a hybrid solver, having part of the model operated on by CP inference and part of the model constituting an LP relaxation. We propose mixed (global) constraints spanning both discrete and continuous variables to bridge the gap between CP and LP, providing inference as well as information for search strategies. Inference comes as cutting planes and domains reductions for LP and CP respectively. Branching is done in the CP part, utilizing information from the LP relaxation.

Apart from a general framework for integration, specific constraints and structures studied include several variations of variable subscripts and piecewise linear functions. Computational experiments show the benefit and potential of a hybrid scheme, illustrated on production planning and configuration problems.

Greger Ottosson, Computing Science Department, Information Technology, Uppsala University, Box 311, SE–751 05 Uppsala, Sweden.

ISSN 0283–359X
ISBN 91–506–1396–0
Printed by Nina Tryckeri HB, Uppsala 2000

To my Parents

Acknowledgments

I like to think that this thesis is the result of the meeting and initiated merger of two fields with different principles, practices and almost religious beliefs. This has made my task inspiring and exciting, but it would not have been possible without expert guidance from both fields.

I would first of all like to thank my advisor Mats Carlsson for his advice and guidance during the last five years. From the first "constraints" course I took for him, until today, he has been as sharp in his comments, criticism and suggestions as in his famous programming skills.

I am also indebted to my other advisor, John Hooker. I have probably grasped less than 10% of what he knows about logic and optimization, but even so it has been invaluable. Our different backgrounds made this collaboration fruitful, and I am thankful for his guidance.

Many thanks, Erlendur, for working with me during the last couple of years. We're a good team, and I wish you the best of luck.

My appreciation also goes to all the past and present members of the department, including Björn Carlson, Arne Andersson, Håkan Millroth, Jonas Barklund, Kostis Sagonas, Per Mildner, Sven-Olof Nyström, Johan Bevemyr, Tomas Lindgren, Niklas Kaltea, Alexander Bottema, Jan Sjödin, Erik Johansson, Lars Thalmann and others. I will miss these years.

For recent and past collaboration, thanks to Hak-Jin Kim, Björn Carlson, Nicolas Beldicaneau, Tomas Axling, Lise Getoor, Markus Fromherz and Mikael Sjödin.

Hans, Anita, Rikard, Ingela, Fredrik, och Henrik, thanks for being a solid foundation in my life. Family values, the Swedish way. This thesis is for you.

Finally, thankful shouts to all the people in Uppsala, Pittsburgh and elsewhere, who have made the last ten years more than enjoyable: Mic, Annika, Pekka, Ingela, Kiina, Anna, William, Fredrik, Petra, Vanja, Art, Sara, Happi, Gaffe, Karin, Linus, Niklas, Erlendur, Sonja, Jochen, Bianca, Kostis, Vipul, Lise, Mallika, Neil, Vivek and Ulf. I'll be back :).

Last in this long line I hide my loving appreciation of Sarette. Her unrivaled love, beauty and wit lend shine to the end of this list, as well as to my life.

Prior Publications

The papers in this thesis have been published previously or are in the process of being published. They have not been edited, expcept to adjust typography, remove typographical errors and update references.

I. John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson and Hak-Jin Kim. **A scheme for unifying optimization and constraint programming**. *Knowledge Engineering Review, special issue on AI/OR, accepted for publication*, 1999. [85]

II. John N. Hooker, Hak-Jin Kim and Greger Ottosson. **A declarative modeling framework that integrates solution methods**. *Annals of Operations Research, Special Issue on Modeling Languages and Approaches, accepted for publication*, 1998. [82]

III. John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson and Hak-Jin Kim. **On integrating constraint propagation and linear programming for combinatorial optimization**. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 136–141. AAAI, The AAAI Press/The MIT Press, July 1999. [84]

IV. Greger Ottosson, Erlendur S. Thorsteinsson and John N. Hooker. **Mixed global constraints and inference in hybrid CLP–IP solvers**. In *CP99 Post-Conference Workshop on Large Scale Combinatorial Optimisation and Constraints*, October 1999. [108]

V. Greger Ottosson and Erlendur S. Thorsteinsson. **Linear relaxations and reduced-cost based propagation of continuous variable subscripts**. In *CP-AI-OR'00 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, March 2000. [107]

Contents

Summary

In this chapter, we briefly introduce combinatorial optimization, constraint programming and integer programming. We give a perspective on relaxations, inference and search, and summarize the papers in this thesis and our contributions.

1.1 INTRODUCTION

Decision problems are ubiquitous in our daily walk of life, and in areas such as industry, finance and transportation, these problems are recognized as being important or necessary to solve. Many of of these can be formulated as *feasibility* or *optimization problems*, in which the question is to decide values for various variables representing some quantity, time, location or selection. The problems range from determining the amount of stock to purchase (portfolio management), what machine to use for production and when (resource-constrained scheduling), when and where to invest in a new plant and where (production planning and facility location) to what component to include in a product and in what amount (configuration).

In the study of these optimization problems, *combinatorial optimization* [101, 110] deals with those in which discrete choices or disjunctions are present. For example, a choice between using machine A or machine B for a certain product is a disjunction, and other discrete choices arise naturally due to the restriction to produce, sell or buy in units or to allocate in slots. Many problems are a mix of discrete and continuous elements, where some variables take on continuous values governed by the discrete decisions made.

In the Operations Research (OR) community, combinatorial optimization sprung from the advancement in continuous optimization, with its foundation in the rise and advancements of Linear Programming (LP) (Dantzig [39]). This development took place in the second half of the 20th century, and defined the field now known as (Mixed) Integer Programming, (MIP) IP (see e.g. Wolsey [146] and Nemhauser and Wolsey [101]).

More recently, during the 1980s and 1990s, Constraint Logic Programming (CLP) evolved within the Computer Science (CS) and Artificial Intelligence

(AI) community [96, 137, 140]. Without the pervading tradition of using linear programming, the growing CLP community took a different approach to combinatorial optimization, more based on logical inference than linear programming. Although CLP schemes have been defined for several other domains (as instances of the CLP($\mathcal{X}$)-scheme, Jaffar and Lassez [87]), such as rational numbers and sets, we use the term CLP and CP throughout this thesis to refer to *finite domain* constraint programming, sometimes denoted CLP(FD).

A third well-known technique, used within both computer science and operations research, is *local search* alias *neighborhood search* [66, 120]. Based on a heuristic exploration of the search space through the investigation of solutions close (or *neighboring*) to a known solution, this class of algorithms has been applied successfully to many combinatorial optimization problems for decades.

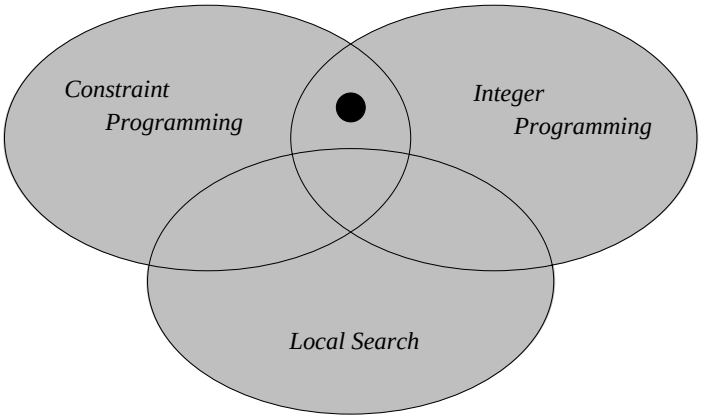


Figure 1.1: Three techniques for combinatorial optimization

Figure 1.1 shows these three techniques and their intersection. The black dot marks the focus of this thesis, i.e. in the intersection of Constraint Programming and Integer Programming.

The rest of this section is is organized as follows. We begin by describing the fundamental building blocks of algorithms for combinatorial optimization; *search*, *relaxations* and *inference*. Next, we discuss some similarities between CP and IP in solution techniques and modeling practices, followed by a summary of the five papers comprising this thesis. We then give some details on the design and implementation of our hybrid solver, before expanding on related work. Finally, we summarize our contributions and conclude with topics for future work.

1.1.1 Search

Search is the process of systematic exploration of a set (or space) of possible solutions. In the context of optimization, the goal of the search is to assign values to variables so as to satisfy given constraints. A search algorithm is said to be *complete* if it is guaranteed (given enough time) to find a solution if it exists, or report lack of solution otherwise.

For combinatorial optimization, most complete algorithms are based on *tree search*, that is, they implicitly define a tree where internal nodes correspond to partial solutions, branches are choices (partitioning the search space), and leaf nodes are complete solutions.

The branch-and-bound algorithm (Land and Doig [95]) is a complete tree-based search used in both CP and IP, although in slightly different shapes. In CP, the branching is combined with inference aimed at reducing the amount of choices needed to explore. In IP, the branching is intertwined with a relaxation, which eliminates the exploration of nodes for which the relaxation is infeasible or worse than the best solution found so far.

Within the class of tree-based search algorithms there exist numerous variations. Classically, one identifies three choices to make in each step; what *node* to continue exploring, what *variable* to branch on and what (set of) *values* to restrict it to. In IP, the *best-bound* node heuristics selects the node with the most promising relaxation value. In CP, the *fail-first* principle [135, 62] chooses the variable with minimum domain, and branches on min-conflict [100]. Recently, schemes tailored at more selective exploration of the tree such as Limited Discrepancy Search (LDS) [73] have gained interest for tackling large-scale problems.

In addition to tree search, there are various schemes of *incomplete* search, of which *neighborhood search* and *genetic algorithms* are the most well known. These iterative procedures maintain one or more complete solutions which are gradually improved by minor changes. These algorithms usually scale better than tree search on hard problems, but they run the risk of getting stuck in local optima and can therefore not guarantee to ever find the globally best solution.

1.1.2 Relaxations

A relaxation of a problem is a formulation that includes and enlarges the set of feasible solutions to the problem. In plain English, it is a simpler problem with additional solutions. The relaxation provides a *lower bound* (assuming minimization) since the optimal solution of the problem cannot be better than the optimal solution of the relaxation. The relaxation also provides a point in space around which the search can be centered and, in case of a

good relaxation, this point is close to the solution of the original problem. This fact is exploited in traditional branch-and-bound search in IP, which branches on fractional values violating the integrality requirements, thus effectively exploring regions close to the relaxation optimum. Branching on inconsistencies is preferable when the relaxation is tight and conforms closely to the original problem, since then the inconsistencies are few and easy to resolve with enumeration.

The relaxation should (to be useful) satisfy two criteria; a) it should be faster to solve to optimality than the original problem, and b) its "solution structure" should resemble the original problem as closely as possible to provide strong bounds.

The most common relaxation is to formulate the original problem in a linear form, drop the integrality requirements and solve it using linear programming. The success of this relies on the efficient LP codes that have been developed and the fact that this relaxation is readily available given that your problem fits in a linear model.

To an extent, the constraint propagation in CP can also be seen as a relaxation, or rather as a process creating a relaxation (the so called *constraint store*). Comprised of the domains of the variables, the constraint store satisfies the first criterium of a good relaxation, but it is uncommon that a set of local constraints provide a globally strong bound. However, some implementations of global constraints [1, 16] utilize discrete relaxations for bounding. As an example, preemptive scheduling (tasks can be interrupted) is a relaxation of non-preemptive scheduling, and can be used as such to provide a bound for the latter problem. Even so, it is not common that this relaxation, or other relaxations in CP, are solved to assign values to variables, which means that the relaxation is not used as a guidance for focusing the search around the optimal solution (as is common practise in IP).

1.1.3 Inference

An *inference* method attempts to derive a desired implication from a set of constraints. Inference occurs in various forms in algorithms for combinatorial optimization. In general, inference *strengthens* the problem by adding valid constraints. Put differently, inference reduces the search space; in CP by reducing domains (adding `indomain` constraints to the constraint store) and in IP by eliminating fractional regions of the polytope by adding cutting planes.

Inference in CP has traditionally been focused on feasibility, where constraint propagation removes only infeasible elements from domains, as opposed to sub-optimal ones. On the contrary, inference in IP has been opti-

mization oriented, e.g. through cutting planes focused around the optimal solution to the relaxation.

Although inference in CP has been developed more systematically with notions of consistency, complexity of propagation algorithms and fix-point loops, inference plays a big role also in IP. Preprocessing that eliminates variables or strengthens constraints (see e.g. Savelsbergh [130]) can be regarded as inference. So can also cutting planes, whether generated up-front or in a branch-and-cut scheme. Even the wide range of decomposition methods (Benders, Lagrangean, Dantzig-Wolfe, etc) can to a large extent be viewed as methods based on inference.

That inference in CP and IP have taken so different forms, is largely due to the underlying solution technologies, constraint propagation versus linear programming.

1.1.4 Analogies in CP and IP

Since the fields of Constraint Programming and Optimization has been developed mainly separated from each other, it is worthwhile to study in what degree similar or equivalent techniques have been invented in parallel. Not only will it help in narrowing the gap between the communities and make it clearer how the different fundaments have influenced the shape of similar techniques, but also perhaps suggest future ideas for tight schemes of integration.

The first analogy that comes to mind is *cutting planes* [146] versus *redundant constraints* [65, 96]. Both serve the purpose of making the underlying solution technique more effective; cutting planes by strengthening the relaxation, and redundant constraints by improving the propagation.

If we look at branching strategies, there are also similarities. Max-regret in CP is a way in which a variable and value is chosen so as to minimize an (estimated) risk, usually the difference between the optimal value choice for a variable and the second-best [32]. For examples of its use see Caseau and Laburthe [34] and Focacci et al [52]. Reduced costs (and estimated pseudo-costs) serve a similar purpose in IP, where a fractional variable is selected according to the expected minimum degradation in relaxation value (see XPress-MP [43] and CPLEX [38]). Reduced costs can actually be used to estimate the regret of variables in a CP framework (Focacci et al [52]).

Recording of nogoods [135] is a well-known technique in the AI and CP community. It is based on discovering infeasible combinations of values or branches while searching and use them to avoid exploring similar regions again. There are ties between this and Benders decomposition [17], which is something we discuss in more detail in Paper I.

Constructive disjunction [147] is a CP technique in which common information of alternatives (disjuncts) are lifted up and posted as constraints. This allows inference on other parts of the model to improve its propagation and for certain classes of problems this improves the overall pruning of the search space in a beneficial way. This is closely related to how *preprocessing* and *probing* (Savelsbergh [130]) are done in IP.

Symmetry elimination is an important part of the art of modeling in CP (see Backofen and Will [4]), and constraints are used for this purpose in each and every other problem formulation. Symmetries are a problem also in IP models, but how they are addressed differs somewhat since the symmetry elimination constraints used in CP sometimes are more complicated to express on linear models. However, Nemhauser and Wolsey [101] and Williams [144] discuss symmetries in an IP context.

1.1.5 Modeling and Software Practices

While comparing the solution approaches of constraint programming and operations research is necessary for an integration, it is also interesting and instructive to consider the different practices of modeling and design of solvers. This could be called the software practices of optimization, in a broad sense denoting how solvers and modeling languages are designed and implemented to interact.

In integer programming, the underlying linear form is historically a natural part of the solution process. This very restricted form of problem representation has permeated also the higher levels of problem formalization and modeling. The lack of high-level modeling constructs for disjunctions and combinatorial constraints makes the process of modeling harder and more error-prone. In addition, much of the problem structure is lost before the problem reaches the solver algorithms.

Furthermore, the black-box structure of IP solvers that are common in OR does not permit flexible, problem-specific search strategies to be defined. This is most probably due to the fact that optimization grew out of the applied mathematics community rather than computer science.

In comparison, constraint programming is influenced more from computer science and programming language design than from mathematics. This has affected not only solution strategies, but also the interface and modeling practises. A wider range of symbolic (global) constraints are available and the user can define new ones in various ways [30, 60]. The flexible modeling practices of CP have encouraged its application to a wide range of diverse and perhaps even odd areas (see e.g. [94, 106, 109]).

However, CLP models are traditionally not as syntactically clear and concise as those written in an algebraic modeling language. This introduces

a unnecessary level of complexity for the human modeler, especially the novice one.

At the same time as solution technologies of CP and IP merge, so will modeling practises. Algebraic modeling will become more flexible, allowing problem-specific preprocessing, inference and search to be defined. This is not only important in the modeling phase, but the model is tightly linked to the solution strategies employed, which is one of the points we argue in this thesis.

1.2 THIS THESIS

This thesis is based on the synthesis of the following beliefs:

- That relaxations are important for bounding and for guidance of the search.
- That inference is essential in reducing the search space and to strengthen the relaxation.
- That relaxations, search and inference should be tightly linked.
- That modeling practises go hand in hand with solution techniques, and that hybrid solvers require a rethinking of our modeling languages.
- That structure identified in the modeling phase should be preserved to the solving phase; partly by modeling using global constraints.

1.2.1 Paper I: A Scheme for Unifying Optimization and Constraint Satisfaction Methods

Although developed in different communities with different perspectives, Constraint Programming and Integer Programming share the same fundamental principles; *search*, *inference*, *strengthening* and *relaxation*. This paper shows the interplay between these techniques as they occur in CP and IP, and delineates two basic dualities – the duality of search and inference, and that of strengthening and relaxation.

The search/inference duality is evident in branching-based search algorithms. During the branching process, one can generate inferences in the form of cutting planes (as in optimization) or constraint propagation to achieve domain reduction (as in constraint satisfaction). The combination of branching and inference is usually much more effective than either alone.

The strengthening/relaxation duality is also evident in branching algorithms. When one branches on the possible values of a variable, the resulting subproblems are strengthenings of the original problem in the sense

of a restriction; they have an additional constraint that fixes the value of the variable and therefore shrinks the feasible set. This strengthening is then relaxed before solved to optimality as a subproblem. Again enumeration of strengthenings is more effective when combined with relaxation of some kind.

The thesis of this paper is that because both constraint programming and optimization use problem-solving strategies based on the same dualities, their methods can be naturally combined. Rather than to employ optimization methods exclusively or constraint satisfaction methods exclusively, one can focus on how these dualities can be exploited in a given problem class.

As an illustration of this, domain reduction rules are shown together with a linear relaxation for variable subscripts (the `element` constraint).

Scientific contributions: By identifying the underlying principles of CP and IP, the paper clarifies their relation and identifies a path to their merger. More generally, it opens up for various ways to combine the principles of inference, search and relaxations, some of which are yet to be explored.

Author's contributions: This paper was largely a joint effort of the authors, except for the part on inference duality, nogoods and Benders (Sections 2.3.2–2.3.3), and relaxation duality (Section 2.4.1), which is based on work of John N. Hooker.

1.2.2 Paper II: A Declarative Modeling Framework that Integrates Solution Methods

This paper explores the relationship between modeling practices and solution techniques for the purpose of identifying how the modeling language can help to integrate different solution techniques.

It is undoubtly the case that solution technology and modeling languages go hand in hand. For example, the underlying solution technology has strongly influenced the modeling framework of mathematical programming. Inequality constraints, for example, are ubiquitous not only because they are useful for problem formulation, but because solvers can deal with them. Linear programming is tethered to a highly structured modeling language – with its limitations – because the solver requires it. Similarly, the open design and richer modeling resources in CP languages (as extensions to general programming languages) have their roots in the tradition of problem-specific search and tailor-made constraint propagation of high-level constraints.

The important lesson is that the modeling language should guide the human modeler to formalize the problem in way that the underlying solution technologies can handle effectively. In this paper, the requirements of a modeling language and framework for combining inference techniques and optimiza-

tion are worked out. In generality, it is shown what conditions should be met for such an integration to produce algorithms which are complete. Possible techniques that can be integrated include interval solvers and non-linear solvers, along with constraint propagation and resolution.

Specifically, Mixed Logical/Linear Programming (MLLP) is introduced as a framework for combining inference (in form of constraint propagation) and optimization (in form of linear programming).

Scientific contributions: The paper gives general conditions for completeness of search that unite different solution strategies. A framework, MLLP, is proposed specifically for integration of constraint propagation and linear programming.

Author's contributions: This paper was a joint effort of the authors.

1.2.3 Paper III: On Integrating Constraint Propagation and Linear Programming for Combinatorial Optimization

Building on the general ideas in Papers I and II, the focus is now specifically on constraint propagation as inference and linear programming as optimization, and how they can be combined. This paper studies the characteristics of these two techniques, and shows how they can be integrated in a natural way where the basic link between inference and optimization is a conditional constraint.

This conditional structure is a key idea in MLLP, which is proposed and expanded on as a framework in which CP and IP can be merged. A multi-machine scheduling problem is used to illustrate how a relaxation partially representing the problem can be combined with constraint propagation.

Furthermore, solution techniques for the framework are presented, which lay the basis for the two subsequent papers in this thesis. This includes a first attempt at a linear relaxation for continuous variable subscripts, although this is significantly refined in Papers I and V. It is also shown how nogoods can be formed in this framework for nodes where the LP subproblem is infeasible, and also how feasible LP solutions can be extended to complete MLLP solutions by solving a satisfiability problem. This is the embryo of the *back-propagation* technique presented in Paper IV (see Section 5.4.1).

Scientific contributions: This paper takes another step from general ideas of integration of inference and optimization to a working, practical hybrid CP-IP system. Specifically, communication between CP and LP is built around the conditional constraints, and solution techniques within a branching framework are presented.

Author's contributions: This paper originated in discussions among the authors and in a workshop paper [105] which I initiated, and was later a joint effort of the authors.

1.2.4 Paper IV: Mixed Global Constraints and Inference in Hybrid CLP–IP Solvers

Chronologically, up until this point the work on CP-IP hybrids — and MLLP in particular — had very much followed in the same path as Constraint Logic Programming originally did. In MLLP, the basic constituents were identified (conditional constraints linking CP and LP) and the basic solution methods designed (branching on discrete variables, bounding by LP-relaxation).

The next step which the CLP development took about ten years ago was to introduce global constraints, such as `alldifferent`, `diffn` and `cumulative`. One reason they were introduced is that they allow the modeler to represent a problem in a more "natural" and compact manner, i.e., they extend the expressiveness of the modeling language. But perhaps more importantly, they open up the possibility to include structure specific propagation into a general solver and are thus extremely important for the efficiency of the solver.

The same is true in a framework such as MLLP, where *mixed* global constraints serve both as a modeling tool and a way to exploit structure in the solution process. A 'mixed' constraint is a constraint over both discrete and continuous variables. The conditional constraint is the most basic mixed constraint to which other mixed constraints can be decomposed, analogous to global constraints in CP. And similarly, mixed global constraints improve the solution process by improving the propagation and cutting plane generation. This is illustrated in this paper with mixed global constraints for variable subscripts, disjunctions and piecewise linear functions.

Another important step for MLLP at this point was to design effective search strategies and to increase the information flow (inference) from the LP solution to the CP constraint store. These two goals go hand in hand. An inference process called *back-propagation* is introduced, which through the mixed constraints identify values for the discrete variables satisfying the LP solution. This serves the purpose of eliminating some unnecessary branching, and as a side effect it provides the information needed to define branching strategies in MLLP that are optimization-oriented. A discrete variable for which back-propagation cannot find a satisfying value corresponds to an inconsistency between the LP solution and the discrete constraints, and identifies a way to branch on that inconsistency. This is similar to traditional IP branching — i.e. branching on fractional values.

Computational results for a production planning problem with piecewise linear functions and variable subscripts show that MLLP is competitive with (outperforms) IP for proving optimality (finding the optimal solution).

Scientific contributions: This paper proposes mixed global constraints as the connecting force between CP and LP. These constraints have both relaxation and constraint propagation algorithms. Furthermore, a scheme for bidirectional inference between CP and LP through the mixed constraints is introduced, and new search strategies for hybrid solvers are presented.

Author's contributions: The ideas and techniques described here were developed in close collaboration with Erlendur S. Thorsteinsson and John N. Hooker. I took initiative to this paper and wrote the draft. The 'back-propagation' technique was initially my idea and I wrote the first implementation. Erlendur implemented the piecewise linear relaxation. In other parts this was a collective work with Erlendur and John.

1.2.5 Paper V: Linear Relaxations and Reduced-Cost Based Propagation of Continuous Variable Subscripts

One class of configuration problems is where there are certain components, each from a set of possible types, and where the objective is to find a feasible configuration with a type and quantity of each component so as to optimally satisfy some objective criteria. Part of a model for this problem includes a variant of variable subscripts, and this paper is focused on this mixed (global) constraint.

It is shown how this subscript structure is formulated in typical CP and IP models, and how a strong linear relaxation can take part in a hybrid model. In addition, it is shown how the reduced-costs in the linear relaxation can be used for domain reduction. This effectively becomes another type of inference method from LP to CP, adding to the effectiveness of the hybrid solver by complementing the propagation, relaxations and back-propagation described earlier.

Scientific contributions: This paper presents linear relaxations and reduced-cost based propagation for two variants of variable subscripts. Computational evidence is given that a hybrid CP-IP approach can improve the solution speed on a class of configuration problems compared to both pure CP and IP.

Author's contributions: This work was done in close collaboration with Erlendur S. Thorsteinsson. I initially modeled the problem, designed and implemented the first versions of the reduced-cost based propagation, which was then refined and benchmarked together with Erlendur. I wrote the draft of this paper which was then completed together with Erlendur who contributed the section on reduced costs.

1.3 MLLP - DESIGN AND IMPLEMENTATION

The MLLP framework has been implemented and used for the benchmarks in Papers IV and V. The implementation consists of two mayor components, a *modeling language* and *solver* (see Figure 1.2). The algebraic modeling

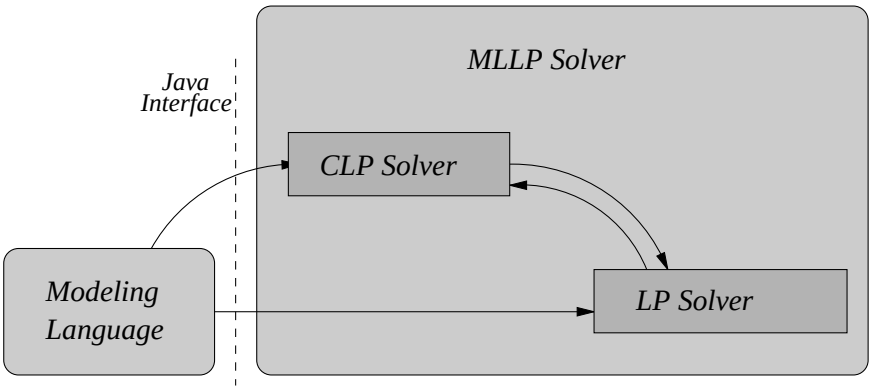


Figure 1.2: The design and implementation of the MLLP framework

language has the same style and syntax as AMPL [59], to a large extent, although it has been extended with

- symbolic constraints, such as `piecewise` and `alldifferent`,
- mixed constraints, such as the `conditional` constraint and variable subscripts in linear inequalities, and
- a language-extension in which problem-specific search can be specified, partly using built-in primitives.

When applied to a model, the modeler component compiles the model and passes it through an interface to the solver. The compilation consists of some basic preprocessing and simplification, but it also compiles variable subscripts into variants of the mixed `element` constraint, as described in Papers IV and V. Unconditional linear inequalities are passed directly to the LP solver, while discrete and mixed constraints are passed to the CP part, since they are part of propagation loops, etc. The modeling language, including parsing and compiling, is entirely written in Java. Example MLLP models for the problems presented in Papers IV and V can be found in Appendix A and B, respectively.

The solver is basically comprised of two parts; a CP part and an LP part. For the latter we use CPLEX LP-solver through its callable library. The CP

part is written in Java and consists of algorithms for constraint propagation and cutting plane generation in mixed constraints (`element` and `piecewise`, for example) as well as search and branching algorithms.

Author's contributions: I implemented the modeler, and the interface to the solver. The initial interface between Java and CPLEX was written by Hak-Jin Kim and later extended by Erlendur S. Thorsteinsson and me. The solver code was initiated by Hak-Jin, and then largely extended and re-written by Erlendur and me, including most or all of the constraint propagation, branching, back-propagation and mixed constraints.

1.4 RELATED WORK

In this section we take a look at other research in this area, beginning with comparisons of CP and IP before we turn to different approaches of integrated and combined techniques.

1.4.1 Comparisons between IP and CP

When aiming towards an integration of techniques from different fields, it is of course important to start with establishing a knowledge base of how the techniques differ, are related and how they may work together.

Several papers compare CP and (M)IP. Darby-Dowman et al [42] apply CP and IP to a set of real-world manufacturing assignment problems. The problem is a slight modification of the Generalized Assignment Problem (GAP), and models for both IP (using standard 0-1 x_{ij} variables) and CP (using integer x_i variables and `element/3` constraints) are introduced and discussed. This paper is interesting for several reasons. Dowman et al. give a fairly detailed description of the differences of the models and how the search tree are affected by those differences. The CP formulation is found to be closer to the structure of the problem which makes it easier to find and apply search heuristics and redundant constraints. Furthermore, the mechanism to control the search is more powerful in the CP solver, which turns out to be valuable. For this problem CP outperforms IP both in CPU time and robustness. The IP solver mainly takes longer time not to find the optimal solution, but to prove its optimality, and also shows unpredictable behavior. The paper also discusses and illustrates on a few examples how CP could be integrated with IP to increase the performance of the IP solver, but this is not explored in generality.

Another well-studied example is the progressive party problem (Smith et al. [132]). This problem is a quite clear case when the linearity requirements of IP force a very large and weak relaxation. In comparison, CP has a more compact model where variables have a more direct mapping to the original problem decisions. The latter allows better search strategies to

be implemented, which in combination with the smaller model makes CP perform better on this problem. This problem has also been studied by Hooker and Osorio [81], and a CHIP model using global constraints is given by Kay [91]. Walser [142] approaches this problem using local search.

In [41], Darby-Dowman and Little expand their investigations and add three more problems for study. The new problems are a golf scheduling problems which bears similarities with the Progressive Party Problem [132]; a Crew Scheduling problem which in essence is a set partitioning problem; and finally a Flow Aggregation problem which is another variant of the GAP called Covering Assignment Problem (CAP) [56]. The results of the experiments confirm some of the previous acclaimed properties of the techniques. The golf scheduling problem is highly constrained, with an awkward IP formulation with many variables and a weak linear relaxation, which makes this very hard to solve with IP. In contrast, the CP formulation is more compact using the constraint `atmost/3`, and CP solves the problem in a few seconds using a customized search procedure. The opposite is true for the crew scheduling problem, which is solved easily using IP, and on which CP cannot find any good solutions. The Covering Assignment Problem is also solved relatively easily with an IP solver, while the CP solver only produces sub-optimal solutions in the same time. Noteworthy is that for the last CP model, a specialized scheduling constraint such as `cumulative/4` [1] is not used, and the same is true for the previous CP model for the GAP, as well as for the golf scheduling problem.

There are several more papers comparing CP and IP. Proll and Smith study a template design problem [116] and Jordan and Drexl [89] compare on a batch sequencing problem with sequence-dependent setup-times. Little and Darby-Dowman [40] describe CP for an OR audience, and compare with IP.

1.4.2 Modeling in CP and IP

In CP, modeling has traditionally been done within a programming language. This is powerful, but not always as concise and clear as with an algebraic modeling language. Barth and Bockmayr [11] show that the basic functionality of algebraic modeling languages can be realized in a CLP language.

Integer Programming has gone from matrix generators to algebraic modeling languages as the way to specify the problem. Examples of modeling languages are AMPL [59], GAMS [24] and XPRESS-MP [43]. Fourer [57] propose an extension of AMPL that incorporates modeling devices from constraint programming, but it does not address the issue of how solvers might cooperate to handle these extensions.

A recent algebraic modeling system, OPL, invokes both linear programming (ILOG Planner/CPLEX) and constraint programming (ILOG Solver) solvers [138]. In OPL it is allowed to use both CP constraints and a linear relaxation, although the interface allows only limited forms of interaction. Links between CP and LP are one-to-one mappings of variables. Problem-specific search can be defined, overriding the default.

We propose in Paper II that the construction of hybrid solvers also requires a rethinking of the modeling process. This is in line with Gervet [64], where it is argued that the new language features made available in a hybrid solver set requirements upstream the solving phase to revise the way the algebraic model and search are formulated, and how the methods are mapped to the model. In particular, there is a need to identify decompositions of the problem so as to map sub-parts of the problem to existing specialized (OR) algorithms.

1.4.3 General Solver Approaches to Combine CP and IP

Earlier work on the MLLP framework is described in Hooker and Osorio [81]. It differs from the current framework on several points, e.g. the framework and models in [81] contained no variable subscripts or (mixed) global constraints (Paper IV), and had a more IP flavor than the MLLP models described in Papers IV and V. The search and inference are now more CP influenced, and schemes such as nogoods (Paper III) and reduced-cost based propagation (Paper V) have been added.

Several other attempts of integration of CP and IP have been made. Early out was Beringer and De Backer [18] in which the idea is explored of coupling CP and LP solvers with bounds propagation and fixed variables. Rodošek, Wallace and Hajian [127], use CP along with LP relaxations in a single search tree to prune domains and establish bounds. A node can fail either because propagation produces an empty domain, or the LP relaxation is infeasible or has an optimal value that is worse than the value of the optimal solution (cutoff). A systematic procedure is used to create a "shadow" MIP model for the original CP model. It includes reified arithmetic constraints (which produce big-M constraints) and `alldifferent` constraints. The modeler may annotate constraints to indicate which solver should handle them — CP, LP or both.

These techniques have since become standard in hybrid branch-and-bound frameworks, i.e. communicating bounds, constraint propagation followed by solving an LP and backtracking on infeasibility (in CP or LP) or cutoff. The linear relaxation in a hybrid model can also be strengthened by adding cutting planes like in pure IP, and in addition the cutting planes can also be operated on by consistency techniques [29, 85]. Furthermore, tighter LP-relaxations can sometimes be accomplished if the linear relaxation of a

symbolic constraint is dynamically rewritten upon domain changes. Refalo [121] proposes a scheme called *tight cooperation* which does this, and illustrates it on piecewise linear functions. Refalo's work on piecewise linear functions is similar to some of ours, although his linearization is different and his branching scheme not as general as what we accomplish through our back-propagation (Paper IV).

Bockmayr and Kasper propose an nice framework in [20] for combining CP and IP, in which several approaches to integration or synergy are possible. They investigate how symbolic constraints can be incorporated into IP much as cutting planes are. They also show how a linear system of inequalities can be used in CP by incorporating it as a symbolic constraint. They also discuss a closer integration in which both linear inequalities and domains appear in the same constraint store.

Carlsson and Ottosson [29] compare CP, IP and a hybrid algorithm for a configuration problem, which in its basic form is a fairly clean linear integer problem, but has extensions which pose a potential problem for a pure IP approach. Linear relaxations, branch-and-bound search, cutting planes, preprocessing [130] and bounds-consistency [96] are experimented with in both pure and mixed variants. Computational experiments show that for this problem linear relaxations and cutting planes are the most important factors for efficiency and that preprocessing and bounds-consistency help on this problem, but are significantly less important than a tight relaxation.

There are also approaches where the CP and IP models are not merged. Heipcke [74] proposes a scheme where two different models (CP and IP) are solved separately in two synchronized search trees. The two models are linked by variables, which provide communication of information between the solvers.

Jain and Grossmann [88] present a scheme where the problem is decomposed into two sub-parts, one handled by IP and one by CP. This is exemplified with a multi-machine scheduling problem where the assignment of tasks to machines is formalized as an IP, and the sequencing of the tasks on the assigned machines is handled with CP. The implemented search scheme (of several presented) is an iterative procedure, where first the assignment problem is solved to optimality (identifying which machine to use for each task), and then a CP feasibility problem is solved trying to sequence according to this assignment. If the sequencing fails, cutting planes are added to the IP problem to forbid this (and subsumed assignments) and the process is iterated.

Constraint programming solvers can for some problems be more efficient for finding a *satisfying* solution, but not perform as well as MIP for finding and proving optimality, and vice versa. Hajian et al [70] describe how the ECL^{i}PS^e (finite domain) constraint system is used for finding a good

feasible initial solution to a fleet assignment problem which is then used to "warm-start" a traditional branch-and-bound IP solver which on its own had problem finding a starting point.

1.4.4 CP Extensions in IP

Some research has been aimed at incorporating better support for symbolic constraints and logic in IP. Hajian, Rodošek and Richards [71, 72] show how dis-equalities ($X_i \neq X_j$) can be handled (more) efficiently in IP solvers. Furthermore, they give a linear modeling of the `alldifferent` constraint. This is done by introducing *non-zero* variables, i.e. variables which cannot take zero as value. This restriction is not expressed as constraints in the model, but rather handled implicitly through an extension to the branch-and-bound algorithm. This is similar to how SOS variables are usually treated.

How to express logical relationships in 0-1 (in)equalities is a fairly well-studied problem; not only `and`, `or`, but also `exactly` and `atmost`. This is described in e.g. McKinnon and Williams [97] and Williams [143, 145], and a tool is presented by Hadjiconsantinou et al in [69]. These transformations can of course also be used for complementing logical constraints in a CP solver with their representation in an LP solver.

Many schemes for logic-based branching in IP can be seen as a step towards CP, whether it was the source of inspiration or not. An interesting example is specially ordered sets (SOS) variables (originally introduced by Beale and Tomlin [13]; see also de Farias [48]), where special instances of cardinality constraints are maintained implicitly in the search instead of explicitly being part of the linear inequalities. There are similarities between this branching and how branching is carried out in CP. For example, branching on a general integer variable in a CP model which corresponds to a set of 0-1 variables in the IP model is equivalent to how branching is commonly done on SOS type I variables. Note that in many cases the 0-1 variables are neither needed nor introduced in a hybrid CP-IP model (Paper V), not even to accomplish this style of branching.

1.4.5 OR Algorithms in CP

Apart from the various optimization schemes centered around (Mixed) Integer Programming, the Operations Research community has developed a wide range of algorithms for specific problems, such as scheduling [3, 26, 27] and network flows [2] to name a few. The integration of these algorithms, along with graph algorithms from applied discrete mathematics, is by far the biggest integration effort in CP. These algorithms are encapsulated in global constraints (see for example [7, 117]), and range in application areas from scheduling [31, 32, 102] and routing [34] to allocation and packing

[123, 1, 16, 104, 28]. For a classification scheme of global constraints, old and new, see Beldicaneanu [15].

The *reduced costs* of a relaxation have long been used in OR for inference (strengthening of bounds). A line of research by Focacci, Lodi and Milano [52, 53, 54, 55], use the reduced costs of an assignment subproblem for propagation in a CP framework, namely within ILOG Solver's `alldifferent` and `path` constraints. The importance of this approach lies in the fact that the propagation in these constraints now becomes *optimization-oriented*, i.e. not only infeasible elements are removed from domains, but also *suboptimal* ones. This partly remedies the lack of relaxations in CP.

Another line of research by Bockmayr and Barth [10, 22, 9] instantiates the CLP($\mathcal{X}$)-scheme to pseudo-boolean constraints, i.e. 0-1 IP problems. Cutting plane techniques from IP provide improved consistency within the CLP paradigm.

1.4.6 Computational Studies of Hybrid Solvers

There is a quite large body of evidence that a hybrid CP-IP approaches and logic-based extensions to IP can bring both modeling and algorithmic advantages. It includes computational studies involving chemical process design [118, 136], distillation network design [67, 81, 119], truss structure design [23], machine scheduling [74, 88], scheduling with resource constraints [114], highly combinatorial scheduling [81, 127], dynamic scheduling [50], production planning and transportation with piecewise linear functions [108, 121], warehouse location [20, 138], traveling salesman problem with time windows [54], hoist scheduling [126], maintenance scheduling of power plants [68] and problems with specially ordered sets [48].

1.4.7 Other Hybrid Approaches

Integer and Linear Programming are not the only techniques which have been integrated with CP. Two other directions have recently also gained a lot of interest. The first is the combination of CP and *local search*. The combination is attractive because local search scales well for handling real-world sized problems, while CP is better suited for problem representation and consistency maintenance. Shaw [131] proposes a technique called *large neighborhood search* in which CP is used *within* the neighborhood search as a way to find a new neighboring solution. This is essentially a combination of *relaxation* and *(re-)optimization*, in which a solution is relaxed by removing the values of a part of the variables of the current solution (keeping some), and then optimizing the objective within the search space just opened up. Using a relatively costly method like CP for this purpose has the benefit

that larger and more coherent moves can be taken in each iteration of the neighborhood search. Related approaches have been explored by Pesant and Gendreau [112, 113], Kilby, Prosser and Shaw [92] and de Backer et al [47].

The second recent advance in integration is that of combining CP with *genetic algorithms*. Various models for integration are possible (Stuckey and Tam [133]), but broadly the possibilities include the following. The complete tree-search commonly used in CP can be replaced by a (hopefully) more efficient incomplete evolutionary programming, in which *chromosomes* are maintained, and evolved using crossover and mutation (Michalewicz [98]). The benefits of using CP for the problem representation are at least two-fold; chromosomes can now be a set of variables with *domains* instead of ground values which allows them to cover a larger part of the search space (Andino and Ruz [129, 128]), and secondly, constraint propagation takes part in the evolutionary search by triggering and propagating in the generation of new chromosomes through crossover and mutation. This is similar to how CP is being used in large neighborhood search discussed above. A slightly more de-coupled integration of CP and evolutionary search is to first use CP to generate constraints, which are then solved using a genetic algorithm (Kok et al [93]).

Local search methods are also growing closer to IP and CP. Walser [142] presents a generalization of local search methods for propositional satisfiability to linear integer optimization. The technique is shown to be efficient and scalable, and problems are defined by an algebraic model.

Column generation [146, 8] is an Integer Programming approach for large scale problems, where variables (columns) representing partial solutions are generated dynamically into a set partitioning IP problem (see e.g. [101]). The success of this approach comes partly from the fact that the solution of the IP problem indicates how new columns should be generated and when optimality has been found. Column generation has been used for cutting-stock problems [141] and crew-scheduling for the airline industry [8]. Recently, CP has been introduced as a method for generating the columns (Junker et al [90]), which has been shown to be effective in expressing the many complicated constraints arising in airline scheduling due to e.g. labor rules and union regulations. This use of CP in combination with IP column generation is similar to the hybrid approach of Clements et al [36], where a local search heuristic for scheduling is used to generate the columns. The problem is a heterogeneous multi-machine scheduling problem, where the columns represent a schedule for a single machine. The set partitioning problem allows for recombination of the schedules generated by the local search procedure, which is shown to improve the quality of the solutions.

It is likely that traditional optimization techniques such as IP, CP and local search will be combined in many more ways in the future. Eventually the border between different search strategies will become blurred, and search will become seamlessly intertwined with both relaxations, inference and other techniques aimed at reducing the search needed.

1.5 CONTRIBUTIONS

The contributions of this thesis can be summarized as follows.

- We have clarified important properties of the underlying solution technologies of CP and IP, and illustrated how they naturally can be combined (Papers I and II).
- We have designed and implemented a complete hybrid CP-IP solver, including modeling language, preprocessor, inference, relaxation and search (Papers II–V).
- We have identified small, effective and dynamic linear relaxations for several important symbolic constraints (Papers IV and V).
- We have introduced the notion of *mixed constraints*, i.e. constraints spanning the border between CP and LP, which allow for easier modeling and more compact linear relaxations (Papers IV and V).
- We have designed a scheme, *back-propagation*, in which information from an LP-solution is passed through these mixed constraints to CP for completion and branching strategies (Paper IV).
- We have proven our scheme to be more efficient for several problems than traditional CP and IP techniques (Papers IV and V).
- Taken together, we have advanced the field of hybrid solvers combining Constraint Programming and Integer Programming by increasing the understanding and designing new algorithms for Combinatorial Optimization.

1.6 CONCLUSION

Since this thesis is on the boundary between OR and AI/CS, I would like to conclude by identifying a few key lessons learnt from our integration efforts. It is my intention for these points to be fairly short-term and practical, rather than too long-term and visionary. For visionary topics, see the end of Paper I, Section 2.6.

1.6.1 Lessons for the OR-community

- Preserve problem structure from modeling phase to solution phase. This includes not linearizing all constraints at first, but using global constraints which the optimization algorithms can give special consideration.
- Use inference (e.g. in the form of constraint propagation) alongside relaxations. Strengthen relaxations by adding cutting planes through systematic inference encapsulated in global constraints.
- Take equal care in hand-crafting problem-specific search as in defining the relaxation. Regard both search and relaxation as part of the model.
- Separate subproblem from relaxation. There can be many subproblems (sub-structures/global constraints), and not all of them need to be relaxed nor treated with inference.

1.6.2 Lessons for the CP-community

- Incorporate relaxations in the CP-framework. This includes but is not limited to LP-relaxations. Discrete relaxations from, for example, graph theory can also be used.
- Focus search around the optimal solution, as indicated by values assigned to variables in the solved relaxation.
- Use the lower bounds given by the relaxation. These are tighter than bounds inferred by constraint propagation and allows for stronger pruning of the search tree.
- Make use of the vast OR-literature on relaxations and how they can be strengthened with cutting planes.

1.7 FUTURE WORK

In this section we outline some specific topics for future work.

1.7.1 MLLP Extensions

The search strategy in MLLP is currently a depth-first branch-and-bound search intertwined with (in order within a node) constraint propagation, optimization by linear programming, reduced-cost based propagation and finally back-propagation to complete the LP solution or obtain branching information (see Paper IV for details). The variable selection strategy proven

most effective so far is branching on inconsistent discrete variables, i.e. variables for which no values satisfy the LP solution. Although MLLP fairly easily finds the optimal solution for the problems tried so far, it is probable that this will not be true in all cases. Extensions that might improve this are, among others, *best-bound* node selection (see e.g. Wolsey [146]) which is standard in pure IP solvers or a restricted branch strategy like Limited Discrepancy Search (LDS) [73] which has proven useful in many CP approaches.

For speeding up the proof-of-optimality of a solution, which currently is relatively more expensive than finding it, we need other approaches. Possible ways of doing this is through optimization-oriented inference, like nogoods or Benders-style cuts. However, nogoods have been implemented in MLLP, but turned out to have no effect on the problems tried so far. This is not too surprising, since nogoods only become effective when there are mistakes made in the search which can be identified and avoided in the future; thus, if the search heuristic makes few or no mistakes in aiming towards optimality, no effective nogoods will be found. In fact, experimental studies showed that generated nogoods consistently involved all or almost all of the variables branched on up until the current node. Such a nogood basically says: "Don't return to this node again", which of course is a statement of only academic interest. But, if general nogoods or cuts fail, problem-specific cuts like those used for multi-machine scheduling by Jain and Grossmann [88] could significantly improve the proof-of-optimality and should be investigated.

Finally, more problems need to be experimented with computationally, and more mixed global constraints should be identified and implemented. The production planning problem and configuration problems (Papers IV and V) are not heavy employers of inference, at the discrete side, and the tight integration between relaxation and inference that MLLP provides would be interesting to apply to a scheduling or complex allocation problem.

1.7.2 Multiple Relaxations

In Constraint Programming, combining the inference on multiple problem sub-structures (encapsulated in global constraints) is done by communication through domains. The combination of multiple *relaxations* (and corresponding solvers) is still largely an open question. Integer Programming has focused on a single (linear) relaxation, and in CP relaxations (where they exist at all) are rarely used effectively for bounding and search guidance.

The question is under which circumstances we can apply two separate relaxations on the same problem, and still be able to efficiently bound the objective and guide the search using the relaxation values.

Example 1 Assume we have a problem such as

$$\begin{array}{ll} \min & cx + d(L \Leftrightarrow y) \\ \text{s.t.} & \texttt{alldifferent}(x_1, \ldots, x_n) \\ & \texttt{cumulative}(y_1, \ldots, y_m, D_1, \ldots, D_m, R_1, \ldots, R_m, \mathit{Cap}) \\ & C(x, y) \end{array}$$

where D, L, R and Cap are constant vectors, and that $C(x, y)$ is a set of additional constraints. (This could be the basic part of a minimize-tardiness scheduling problem with some side-constraints.) Assume further that `alldifferent` has a relaxation, the assignment problem, which has been solved to minimize cx with values $(x_1, \ldots, x_n) = (\bar{x}_1, \ldots, \bar{x}_n)$ and that in conjunction with the inference algorithm for `cumulative`, a relaxation has given values $(y_1, \ldots, y_n) = (\bar{y}_1, \ldots, \bar{y}_n)$ which minimizes $d(L \Leftrightarrow y)$. Since variables x and y are separated in the subproblems, if $C(\bar{x}, \bar{y})$ is satisfied we have an optimal solution to the problem. □

Clearly, for problem that decompose into separate sub-parts, multiple relaxations will work. The back-propagation scheme (Paper IV) will pinpoint the culprit if $C(\bar{x}, \bar{y})$ is not satisfied, which makes it possible to branch on the incompatible values. For problems where the sub-parts are more tightly coupled, it is an open question how the relaxations can be combined.

A Scheme for Unifying Optimization and Constraint Satisfaction Methods

John Hooker, Greger Ottosson, Erlendur S. Thorsteinsson and Hak-Jin Kim

Optimization and constraint satisfaction methods are complementary to a large extent, and there has been much recent interest in combining them. Yet no generally accepted principle or scheme for their merger has evolved. We propose a scheme based on two fundamental dualities, the duality of search and inference and the duality of strengthening and relaxation. Optimization as well as constraint satisfaction methods can be seen as exploiting these dualities in their respective ways. Our proposal is that rather than employ either type of method exclusively, one can focus on how these dualities can be exploited in a given problem class. The resulting algorithm is likely to contain elements from both optimization and constraint satisfaction, and perhaps new methods that belong to neither.

2.1 INTRODUCTION

The last several years have seen increasing interest in combining the models and methods of optimization with those of constraint satisfaction. Integration of the two was initially impeded by their different cultural origins, one having developed largely in the operations research community and the other in the computer science and artificial intelligence communities. The advantages of merger, however, are rapidly overcoming this barrier.

There is a growing body of evidence that a hybrid approach can bring both modeling and algorithmic advantages. It includes computational studies involving chemical process design [118, 136], distillation network design [67, 81, 119], truss structure design [23], machine scheduling [74, 88], scheduling with resource constraints [114], highly combinatorial scheduling [81, 127], dynamic scheduling [50], production planning and transportation with piecewise linear functions [108, 121], warehouse location [20, 138], traveling salesman problem with time windows [54], hoist scheduling [126] and problems with specially ordered sets [48]. A recent commercially available modeling system, OPL, invokes both linear programming (ILOG Planner/CPLEX) and constraint programming (ILOG Solver) solvers [138].

Despite these developments, no generally accepted principle or scheme has evolved for the merger of optimization and constraint satisfaction. The purpose here is to propose such a scheme for unifying the solution methods of the two fields. We address elsewhere [81, 82, 84] the issue of a unified modeling framework.

Our scheme is based on exploiting two dualities: the duality of search vs. inference and the duality of strengthening vs. relaxation. Some of these ideas are anticipated in [75].

Branching algorithms provide one example of these dualities at work. The search/inference duality is evident in Bockmayr and Kasper's observation [20] that both optimization and constraint satisfaction rely on "branch and infer." Branching is a search mechanism. During the branching process, one can generate inferences in the form of cutting planes (as in optimization) or constraint propagation to achieve domain reduction (as in constraint satisfaction). The combination of branching and inference is usually much more effective than either alone.

The strengthening/relaxation duality is also evident in branching algorithms. When one branches on the possible values of a variable, the resulting subproblems are strengthenings of the original problem in the sense of a restriction; they have an additional constraint that fixes the value of the variable and therefore shrinks the feasible set. In optimization one typically solves a relaxation of the problem at each node of the search tree in order to obtain bounds on the optimal value, often a continuous relaxation such as a linear programming or Lagrangean relaxation. The reduced variable domains obtained in a constraint satisfaction algorithm in effect represent a relaxation of the problem at that node of the search tree. In any feasible solution the variables must take values in these domains, but an arbitrary selection of values from the domains need not comprise a feasible solution. Again enumeration of strengthenings is more effective when combined with relaxation of some kind.

The thesis of this paper is that because both constraint programming and optimization use problem-solving strategies based on the same dualities, their methods can be naturally combined. Rather than employ optimization methods exclusively or constraint satisfaction methods exclusively, one can focus on the how these dualities can be exploited in a given problem class. The resulting algorithm is likely to contain elements from both optimization and constraint satisfaction, and perhaps new methods that belong to neither.

Section 2.2 begins the paper with a simple illustration of how the dualities might operate in a branching context. It does so first in a constraint satisfaction and in an integer programming setting. It then combines the two approaches.

The next two sections explore the dualities more deeply and propose methods that belong to neither constraint satisfaction nor integer programming. Section 2.3 investigates the search/inference duality. It explains how constraint programmers exploit problem structure to apply effective inference algorithms. It also frames the search/inference duality as a formal optimization duality that generalizes classical linear programming duality. This perspective forges a link between the concept of a nogood in constraint satisfaction and Benders decomposition in optimization. It also provides a general method for sensitivity analysis.

Section 2.4 takes up the duality of strengthening and relaxation. This duality is studied in optimization under the guise of Lagrangean and surrogate duality, which finds a strong relaxation by searching over a parameterized family of relaxations. Both are defined only for inequality constraints, but both are special cases of a general relaxation duality that can be developed for a much wider range of problems.

A maneuver that looks particularly promising is to combine constraint programming's approach to inference with optimization's approach to relaxation. When formulating a problem, the constraint programmer often identifies a group of constraints that show special structure and represents them with a single *global* constraint, such as `all-different`, `element` or `cumulative`. The optimizer often relaxes a problem by transforming it to an instance of a specially structured class of problems that can be solved to optimality, such as a linear programming problem. Both of these techniques are key to the success of the respective fields. There is a natural way to combine them: design relaxations of the sort used in optimization for global constraints of the sort used in constraint programming [108]. This idea is illustrated in Section 2.5 by presenting relaxations for element constraints.

The paper concludes by suggesting issues for future research.

2.2 A MOTIVATING EXAMPLE

A small example can illustrate how dualities can operate in a constraint satisfaction and in an integer programming setting as well as in a combined mode. One example can illustrate only a few of the relevant ideas, but it will help make the discussion to follow more concrete.

Consider the following optimization problem:

$$\begin{array}{ll} \text{minimize} & 4x_1 + 3x_2 + 5x_3 \\ \text{subject to} & 4x_1 + 2x_2 + 4x_3 \geq 17, \\ & \texttt{all-different}\{x_1, x_2, x_3\}, \\ & x_j \in \{1, \ldots, 5\}. \end{array} \tag{2.1}$$

The set $D_j = \{1, \ldots, 5\}$ is the initial *domain* of each of the variables x_j. The optimal solution is $(x_1, x_2, x_3) = (2, 3, 1)$ with optimal value 22.

We will solve the problem by constraint satisfaction methods, by integer programming, and finally by a combined approach. The particular methods illustrated are not the best available for either constraint satisfaction or integer programming. They are chosen because they help illustrate how methods may be combined. The intent is not to compare the performance of constraint programming and integer programming but to present a combined approach.

2.2.1 Constraint Satisfaction

A constraint satisfaction method can solve (2.1) by solving the feasibility problem

$$4x_1 + 3x_2 + 5x_3 < z, \tag{2.2}$$

$$4x_1 + 2x_2 + 4x_3 \geq 17, \tag{2.3}$$

$$\texttt{all-different}\{x_1, x_2, x_3\}, \tag{2.4}$$

$$x_j \in \{1, \ldots, 5\}. \tag{2.5}$$

Initially $z = \infty$. Each time a feasible solution is found, the search continues with z set to the value of the solution. A possible search tree is shown in Table 2.1. The nodes are traversed in the depth-first order shown. At node 1, where $D_1 = D_2 = D_3 = \{1, 2, 3, 4, 5\}$, the search branches on x_1. This creates a subproblem at node 2 by setting $D_1 = \{1\}$, and one at node 7 by setting $D_1 = \{2, 3, 4, 5\}$. (Other branching schemes are possible.) Similarly, the search branches on x_2 at node 2, creating nodes 3 and 6.

Because $x_1 = 1$ at node 2, setting $x_2 = 1$ or $x_3 = 1$ would be inconsistent with the `all-different` constraint. The domains of x_1, x_2 are therefore reduced to $D_1 = D_2 = \{2, 3, 4, 5\}$. There are several different *domain*

Node	D_1 D_2 D_3	z	*Value*	*Branches*
1.	12345 12345 12345	∞	12..60	
2.	1 2345 2345	∞	20..44	$D_1 = \{1\}$
3.	1 2 345	∞	25..35	$D_2 = \{2\}$
4.	1 2 3	∞	25	$D_3 = \{3\}$
5.	1 2	25	∞	$D_3 = \{4, 5\}$
6.	1 3 2	25	23	$D_2 = \{3, 4, 5\}$
7.	23 123 12	23	16..22	$D_1 = \{2, 3, 4, 5\}$
8.	2 3 1	23	22	$D_1 = \{2\}$
9.	3 1	22	∞	$D_1 = \{3, 4, 5\}$

Table 2.1: Solution of a constraint satisfaction problem by branching and domain reduction. The "value" shown is the value or domain of $4x_1 + 3x_2 + 5x_3$ at a leaf node of the search tree. A value of ∞ indicates the lack of a feasible solution.

reduction algorithms that remove domain elements that are inconsistent with `all-different`, varying in degree of efficiency and completeness [96, 123].

A feasible solution is found at node 4 that permits one to set $z = 25$. At node 5 another type of domain reduction, based on maintaining *bounds consistency*, can be applied [96]. It infers from (2.2) that

$$x_3 \leq \tfrac{1}{5}(z \Leftrightarrow 4 \min D_1 \Leftrightarrow 3 \min D_2) \tag{2.6}$$

where $\min D_j$ is the smallest element in D_j, and similarly for x_1 and x_2. This changes neither $D_1 = \{1\}$ nor $D_1 = \{2\}$ but reduces $D_3 = \{4, 5\}$ to the empty set because (2.6) implies that $x_3 \leq 3$. In general, the implications of one constraint are *propagated* to other constraints by means of a *constraint store*, which in the present case consists of the reduced domains that are inferred from one constraint and made available to others.

The subproblem at node 5 is therefore infeasible. Node 6 reveals a solution of value $z = 23$, and finally the optimal value $z = 22$ is discovered at node

8, and the search completes its proof of optimality in 9 nodes. Constraint satisfaction therefore solves (2.1) with a combination of search (branching) and inference (domain reduction).

2.2.2 Integer Programming

Node	$\bar{z}$	*Value*	(x_1, x_2, x_3)	(y_{12}, y_{13}, y_{23})	*Branches*
1.	∞	20	$(3, 1, 1)$	$(0, 0, \frac{1}{5})$	
2.	∞	21	$(2\frac{1}{2}, 2, 1)$	$(\frac{1}{10}, 0, 0)$	$y_{23} = 0$
3.	∞	$21\frac{2}{3}$	$(2, 2\frac{1}{3}, 1\frac{1}{3})$	$(\frac{4}{15}, \frac{1}{15}, 0)$	$x_1 \leq 2$
4.	∞	∞			$x_2 \leq 2$
5.	∞	22	$(2, 3, 1)$	$(1, 0, 0)$	$x_2 \geq 3$
6.	22	23	$(3, 2, 1)$	$(0, 0, 0)$	$x_1 \geq 3$
7.	22	21	$(2, 1, 2)$	$(0, \frac{1}{5}, 1)$	$y_{23} = 1$
8.	22	25	$(3, 1, 2)$	$(0, 0, 1)$	$y_{13} = 0$
9.	22	$21\frac{1}{2}$	$(1\frac{1}{2}, 1, 2\frac{1}{2})$	$(\frac{1}{10}, 1, 1)$	$y_{13} = 1$
10.	22	22	$(1, 1, 3)$	$(\frac{1}{5}, 1, 1)$	$x_1 \leq 1$
11.	22	26	$(2, 1, 3)$	$(0, 1, 1)$	$x_1 \geq 2$

Table 2.2: Solution of an integer programming problem by branching, relaxation, and cutting plane generation.

Integer programming exploits the search/inference duality along with a duality of strengthening and relaxation. Due to the restricted vocabulary of integer programming, however, the model is more complex.

Variables y_{jk} $(j < k)$ are introduced to enforce the `all-different` constraint, with $y_{jk} = 1$ when $x_j < x_k$ and $y_{jk} = 0$ when $x_j > x_k$.

$$\begin{aligned}
\text{minimize} \quad & 4x_1 + 3x_2 + 5x_3 && (2.7)\\
\text{subject to} \quad & 4x_1 + 2x_2 + 4x_3 \geq 17, && (2.8)\\
& x_j \leq (x_k \Leftrightarrow 1) + 5(1 \Leftrightarrow y_{jk}), \quad \forall j, k \text{ with } j < k, && (2.9)\\
& x_k \leq (x_j \Leftrightarrow 1) + 5y_{jk}, \quad \forall j, k \text{ with } j < k, && (2.10)\\
& 1 \leq x_j \leq 5 \text{ and } x_j \text{ integer}, \quad j = 1, 2, 3,\\
& y_{jk} \in \{0, 1\}, \quad \forall j, k \text{ with } j < k.
\end{aligned}$$

Constraints (2.9)–(2.10) illustrate the ubiquitous "big-M" constraint of integer programming; here $M = 5$. When $y_{jk} = 1$, constraint (2.9) is enforced while (2.10) is vacuous, and vice-versa when $y_{jk} = 0$.

There are other and better integer programming models for this particular problem. Model (2.7)–(2.10) is used here in order to illustrate the big-M constraints and how they may be avoided in general.

The problem is again solved by branching. This time, however, a *continuous relaxation* of the problem is solved each node of the search tree. The continuous relaxation of (2.7)–(2.10), which is solved at the root node, is obtained by deleting the integrality constraints on x_j and replacing $y_{jk} \in \{0, 1\}$ with $0 \leq y_{jk} \leq 1$. The optimal value of the resulting linear programming relaxation provides a lower bound on the optimal value of (2.7)–(2.10).

In integer programming, inference often takes the form of *cutting plane* generation. Cutting planes are inequalities that are satisfied by every integer solution of the continuous relaxation but possibly violated by some noninteger solutions. By "cutting off" noninteger solutions, cutting planes can provide a tighter bound when added to the constraint set of the continuous relaxation. In this case one might add the cutting planes,

$$\begin{aligned} x_1 + x_2 + x_3 &\geq 5 \\ 2x_1 + x_2 + 2x_3 &\geq 9 \end{aligned} \tag{2.11}$$

(These cutting planes are derived from the inequality constraints alone.) There is a vast literature describing how cutting planes may be generated for highly structured problem classes, such as traveling salesman, job shop scheduling, set covering, set packing, and a host of other problems.

A search tree for the problem represented by (2.7)–(2.10) and (2.11) appears in Table 2.2. The search branches on variables that have nonintegral values in the continuous relaxation, in the order x_1, x_2, x_3, y_{12}, y_{13}, y_{23}. At node 1, $y_{23} = \frac{1}{5}$, and one branches by setting $y_{23} = 0$ and $y_{23} = 1$, creating nodes 2 and 7. At node 2, $x_1 = 2\frac{1}{2}$, and the branches are defined by $x_1 \leq 2$ and $x_1 \geq 3$. The branching constraints are added to the relaxation at each node. For example, $y_{23} = 0$ is added at node 2. The first feasible (i.e., integral) solution is found at node 5. Its optimal value provides an upper bound $\bar{z} = 22$ on the optimal value of the original problem, i.e., the constraint $4x_1 + 3x_2 + 5x_3 \leq 22$ is added to the problem at node 5. At node 6 the value of the relaxation is 23, so further branching at node 6 cannot lead to an optimal solution, and the tree is pruned at this point. This bounding mechanism provides the name, *branch-and-bound,* for this particular kind of search.

Branch-and-bound search relies on the interplay of the search/inference and strengthening/relaxation dualities. It searches by branching and it draws inferences by cutting plane generation (which results in *branch and cut*). Branching likewise creates strengthenings of the problem by adding constraints. A relaxation of this strengthened problem is created by dropping integrality requirements and applying a linear programming algorithm.

2.2.3 A Combined Approach

Node	D_1 D_2 D_3	$\bar{z}$	*Value*	(x_1, x_2, x_3)	*Branches*
1.	12345 12345 12345	∞	20	$(3, 1, 1)$	
2.	12345 2345 12345	∞	21	$(2\frac{1}{2}, 2, 1)$	$x_2 \geq 2$
3.	12 2345 12345	∞	$21\frac{1}{2}$	$(2, 2, 1\frac{1}{2})$	$x_1 \leq 2$
4.	2 345 1	∞	22	$(2, 3, 1)$	$x_3 \leq 1$
5.	$\varnothing$	22	∞		$x_3 \geq 2$
6.	$\varnothing$	22	∞		$x_1 \geq 3$
7.	$\varnothing$	22	∞		$x_3 \geq 2$

Table 2.3: Solution of a constraint satisfaction problem by branching, domain reduction, relaxation, and cutting plane generation.

The respective advantages of constraint satisfaction and integer programming are easily combined in this case. Domain reduction and cutting plane generation are simply two forms of inference, and both can be used. In fact, domain reduction can be applied to the cutting planes (2.11) as well as the original constraint (2.8). The advantages of relaxation are also available. The integer programming relaxation was obtain by dropping integrality constraints from the large model (2.7)–(2.10). But the largest part of this model, (2.9)–(2.10), adds little to the quality of the relaxation. One can use the original model (2.1) as a problem statement and for feasibility checks, but create a continuous relaxation that consists of (2.7)–(2.8), the cutting planes (2.11) and the bounds $1 \leq x_j \leq 5$. Because the relaxation is distinguished from the model, both are more succinct.

A search tree appears in Table 2.3. At each node constraint propagation is first applied to the original problem, and if successful, the bounds in the relaxation are adjusted accordingly (we add the branching constraints both to the original model and the relaxation). As soon as a feasible solution is found, a constraint is added to the problem indicating the bound on the solution and it is updated as necessary.

The search branches on the alternatives $x_1 \geq 2$, $x_3 \geq 2$ at node 1 because the solution of the relaxation sets $x_2 = x_3 = 1$; the alternatives are obviously exhaustive. At node 2 the search branches on a nonintegral variable x_1. Because the solution of the relaxation at node 3 is integral and satisfies `all-different`, it is feasible, and no further branching is needed.

Due to the combined effects of constraint propagation and relaxation, a combined approach may produce a search tree is smaller than those that result from constraint programming or integer programming methods. In addition processing may be faster at each node than in integer programming, because the relaxation is smaller. There may also be nodes at which one need not solve the relaxation, because constraint propagation alone (which is often faster than solving an LP) may determine that the problem is infeasible.

2.3 DUALITY OF SEARCH AND INFERENCE

A *search* method examines possible values of the variables until an acceptable solution is found. An *inference* method attempts to derive a desired implication from the constraint set. Popular search methods include branching (which examines partial solutions) and local search heuristics (which examine complete solutions). Inference methods include cutting plane methods (in which inequalities are inferred) and domain reduction (in which smaller domains are inferred).

Search and inference tend to work best in combination. Search alone may happen upon a good solution early in the process, but it must examine many other solutions before determining that it is good. Inference alone can rule out whole families of solutions as inferior, but this is not the same as finding a good solution. Working together, search and inference can find and verify good solutions more quickly.

As illustrated above, a common strategy for running search and inference in parallel is to use inference in the context of a branching search. Constraint satisfaction infers smaller domains, for instance by maintaining consistency. Smaller domains result in less branching. The example illustrates the maintenance of bounds consistency for inequalities and hyperarc consistency [96] for `all-different` constraints. The cutting planes of optimization are designed to strengthen continuous relaxations but can reduce domains as well, for example if one maintains bounds consistency for them.

Combining search and inference also provides an effective way to exploit problem structure. Inferences drawn from specially structured constraints can reveal which regions of the solution space are unproductive and need not be examined. This is discussed first below. Finally, the interaction of search and inference can be interpreted as a formal duality. This leads to a link between nogoods and Benders decomposition as well as a general approach to sensitivity analysis.

2.3.1 Inference and Structure

One advantage of using inference in the context of search is that it can exploit special structure. If the problem or some part if it exhibits a pattern

that has been closely analyzed offline in order to identify strong implications, these implications can be generated quickly. The practical success of optimization and constraint satisfaction owes much to this strategem.

The two fields have developed different and complementary approaches to recognizing structure. Constraint programmers identify sets of constraints, at the modeling stage, that can be recognized as a single global constraint (e.g., [16, 32, 34, 122, 124, 125]). The `cumulative` constraint, for example, requires that a set of tasks be scheduled so that, at any given time, their total consumption of resources is within bounds. A variety of scheduling problems are special cases of this general pattern. When they are formulated as such, the solver can apply domain reduction procedures that deal with the constraints *globally* rather than one at a time, thus resulting in much greater reduction of domains.

In optimization, the modeler typically identifies a problem as an instance of a class for which solution methods have been designed, such as linear programming, network flow, or 0–1 programming problems. Beyond this point the recognition of structure is often automated as part of the solution algorithm. The problem is scanned for opportunities to generate knapsack cuts, fixed charge cuts, covering inequalities, etc. In some cases substructures are identified, as for example subgraphs in a traveling salesman problem that give rise to *comb* inequalities. Another difference with the constraint satisfaction community is that constraint generation is aimed at strengthening a continuous relaxation rather than raising the degree of consistency of the constraint set. Interestingly, for a brief period in the early days of operations research, optimizers used constraints that were unrelated to the continuous relaxation. They were part of the *implicit enumeration* schemes of that day (e.g., [61]). Perhaps such constraints have since been neglected because the community, unaware of the theory of consistency, has not had a clear understanding of how they might accelerate search.

One of the most impressive traits of the human mind is its pattern-recognition ability. The constraint satisfaction approach uses this ability to identify structure primarily in the modeling stage. Optimization uses it primarily in the design of the solution algorithms that automatically detect patterns. To restrict oneself to one approach or the other seems a mistake. The modeler's insight into the practical situation should be used, as should the mathematician's analysis when the problem is sufficiently stylized to apply it.

2.3.2 Inference Duality in Optimization

One way to capture the duality of search and inference in a more rigorous setting is to state it as a formal optimization duality. For this purpose a general optimization problem might be written

$$\begin{array}{ll} \underset{x \in D}{\text{minimize}} & f(x) \\ \text{subject to} & x \in S \end{array} \tag{2.12}$$

where S is the feasible set and $D = D_1 \times \ldots \times D_n$ the domain. This can be viewed as a search problem: find an $x \in S \cap D$ that minimizes $f(x)$. The inference dual is

$$\begin{array}{ll} \text{maximize} & z \\ \text{subject to} & (x \in S) \overset{D}{\rightarrow} (f(x) \geq z) \end{array} \tag{2.13}$$

where the arrow indicates implication: for all $x \in D$, if $x \in S$ then $f(x) \geq z$. If an optimal value exists for (2.12), it is the same as the optimal value of (2.13). So optimization can be viewed as an inference problem: what is the tightest lower bound on $f(x)$ one can infer from $x \in S$?

The optimization literature has closely studied inference duality in the special case of linear programming. Here (2.12) becomes

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & cx \\ \text{subject to} & Ax \geq b,\ x \geq 0 \end{array} \tag{2.14}$$

where A is an $m \times n$ matrix. The dual problem is to infer the strongest possible inequality $cx \geq z$ (i.e., the tightest lower bound z) from $Ax \geq b$, $x \geq 0$. A fundamental result of linear programming (the Farkas Lemma) states that if $Ax \geq b$, $x \geq 0$ is feasible, it implies $cx \geq z$ if and only if some nonnegative linear combination $uA \geq ub$ of $Ax \geq b$ dominates $cx \geq z$. That is, $uA \leq c$ and $ub \geq z$ for some $u \geq 0$. So the dual problem can be written

$$\begin{array}{ll} \underset{u \in \mathbb{R}^m}{\text{maximize}} & ub \\ \text{subject to} & uA \leq c,\ u \geq 0 \end{array} \tag{2.15}$$

This is the classical linear programming dual. It has the same (possibly infinite) optimal value z^* as (2.14) unless both (2.14) and (2.15) are infeasible. The solution u of the dual problem can be viewed as encoding a *proof* that $cx \geq z^*$, because it specifies a linear combination of $Ax \geq b$ that dominates $cx \geq z^*$.

Linear programming has the convenient property that a solution of the dual always has polynomial length (i.e., linear programming belongs to both NP and co-NP). A proof of optimality can in general be exponentially long.

For example, if x in (2.14) is restricted to be integer, so that (2.14) becomes an integer programming problem, then the inference dual can no longer be written in the form (2.15). The dual must be solved by a proof of optimality that most commonly takes the form of an exhaustive search tree—which has exponential size in general.

A major benefit of inference duality is that it provides a scheme for sensitivity analysis. This kind of analysis is very important in practice because it indicates how the solution would be affected by perturbations of the problem data. It allows one to focus on data that really matter.

Up to a point, sensitivity analysis is straightforward. Given an optimal solution of the primal problem (2.12), one can analyze under what data alterations this solution remains feasible. But this says nothing about whether it remains optimal, and this is where the inference dual comes into play. Because a solution of the inference dual is a proof, one can analyze under what data perturbations the proof remains valid and the solution therefore remains optimal (assuming it remains feasible as well).

This scheme works out nicely in linear programming. Let x^* be an optimal solution of (2.12) and u^* an optimal solution of the dual problem (2.15). Let (2.14) be perturbed so that it minimizes $(c+\Delta c)x$ subject to $(A+\Delta A)x \geq (b+\Delta b)$. Obviously x^* remains feasible if $\Delta A x^* \geq \Delta b$. But it remains optimal as well if u^* remains a valid proof that $(c+\Delta c)x \geq z^*$; i.e., if $u^*\Delta A \leq \Delta c$ and $u^*\Delta b \geq z^*$.

Until very recently the optimization community has approached the sensitivity question for discrete problems in a different fashion, using the concepts of value function, super-additive duality, etc. These approaches tend to make sensitivity analysis computationally very difficult. Both optimization and constraint satisfaction problems could benefit from the inference duality approach. The branch-and-bound tree for an integer programming problem, for example, can be analyzed to determine under what problem perturbations the tree remains a proof of optimality [44, 76, 77]. If branching and domain reduction prove a problem instance to be infeasible, one can examine for what problem perturbations this proof remains valid.

More generally, a solution of the inference dual provides an *explanation* (in the form of a proof) for why a solution is optimal, or why the problem is infeasible. In practice, an explanation of the solution could be more valuable than the solution itself. Perhaps methods can be developed to reduce this proof to its bare essentials, delineating as clearly as possible the reason for optimality or infeasibility. These ideas remain largely unexplored.

2.3.3 Nogoods and Benders Decomposition

If in the midst of search a trial solution is found to be unsatisfactory, the reasons for its failure can be analyzed. This analysis may lead to a constraint that rules out many solutions that fail for the same reason. By adding this constraint to the problem one can avoid unnecessary search. Such a constraint is a *nogood*, a well-known idea in the constraint satisfaction literature [135]. A nogood is a way of learning from one's mistakes. It combines search and inference in a particular way: the inferred constraints (nogoods) are occasioned by the discovery of bad solutions.

A related idea has evolved in the optimization literature. One way to combine search and inference is to search values of *some* of the variables and use inference to project the constraints onto these variables. Let the variables of (2.12) be partitioned as follows.

$$\begin{array}{ll} \underset{x \in D_x,\, y \in D_y}{\text{minimize}} & f(x,y) \\ \text{subject to} & (x,y) \in S \end{array} \tag{2.16}$$

We will search over values of x. If we examine a particular value $\bar{x}$, we can find optimal values for the remaining variables subject to $x = \bar{x}$. This poses the *subproblem*

$$\begin{array}{ll} \underset{y \in D_y}{\text{minimize}} & f(\bar{x},y) \\ \text{subject to} & (\bar{x},y) \in S \end{array} \tag{2.17}$$

The variables are partitioned in such a way that the subproblem has special structure that makes it easier to solve. The variables may decouple, for example.

The next step is to solve the inference dual of the subproblem by generating a proof of optimality for its optimal solution $y_{\bar{x}}$. This proof might take the form of a branching tree. By examining the conditions under which this proof is valid, we may be able to define a function $B_{\bar{x}}(x)$ that provides a lower bound on the optimal value of (2.16) for a given value of x. Two examples of this are provided below. Obviously $B_{\bar{x}}(\bar{x}) = f(\bar{x}, y_{\bar{x}})$, but even when x has some value other than $\bar{x}$, it may be possible to determine what kind of lower bound the dual proof still provides. If z is the objective function value of (2.16), this analysis yields the nogood $z \geq B_{\bar{x}}(x)$. It states that there is no point in examining solutions x for which the objective function value is less than $B_{\bar{x}}(x)$. The *master problem* minimizes the objective function subject to nogoods that have been accumulated so far:

$$\begin{array}{ll} \underset{x \in D_x}{\text{minimize}} & z \\ \text{subject to} & z \geq B_{x^k}(x), \quad k = 1, \ldots, K \end{array} \tag{2.18}$$

where $x^1, \ldots, x^K$ are the nogoods generated so far. Each time the master problem is solved, the solution $\bar{x}$ generates another nogood. The nogoods in effect project the constraint set onto the variables x. It is usually unnecessary to generate all of the nogoods that define the projection, because the process stops when a nogood is satisfied by the previous $\bar{x}$.

When this strategy is applied to a problem with the following form, the result is *Benders decomposition*.

$$\begin{array}{ll} \underset{x \in D_x,\, y \in \mathbb{R}^n}{\text{minimize}} & f(x) + cy \\ \text{subject to} & g(x) + Ay \geq b \end{array} \tag{2.19}$$

The subproblem is a linear programming problem.

$$\begin{array}{ll} \underset{y \in \mathbb{R}^n}{\text{minimize}} & f(\bar{x}) + cy \\ \text{subject to} & Ay \geq b \Leftrightarrow g(\bar{x}) \end{array} \tag{2.20}$$

The dual solution $u_{\bar{x}}$ of (2.19) proves bound $z \geq u_{\bar{x}} b \Leftrightarrow u_{\bar{x}} g(\bar{x})$. Because the proof remains valid when x has values other than $\bar{x}$, we have the nogood $z \geq B_{\bar{x}}(x) = u_{\bar{x}} b \Leftrightarrow u_{\bar{x}} g(x)$, also known as a *Benders cut*.

When the subproblem (2.17) is solved by branching, one may be able to construct a boolean formula $P(x)$ such that the branching tree remains a proof of optimality of $y_{\bar{x}}$ whenever $P(x) = 1$. It is shown in [83], for example, that in integer programming the branching proof can be viewed as encoding a resolution proof whose premises are implied by constraints that are violated at the leaf nodes of the tree. The violated constraints imply the premises when $x = \bar{x}$. One can let $P(x) = 1$ for all values of x for which these constraint continue to imply the premises. Then $z \geq B_{\bar{x}}(x) = f(\bar{x}, y_{\bar{x}})P(x)$ is a valid nogood (if $z \geq 0$).

Nogoods can be combined with branching. Suppose that at a given node of the branching tree, constraint propagation or cutting planes prove infeasibility, or more generally prove $z \geq \underline{z}$ where $\underline{z} = \infty$ in the case of infeasibility. One can view this proof as solution of the inference dual of a subproblem containing the variables that have not yet been fixed at that node. Then one might derive a nogood $z \geq B_{\bar{x}}(x)$, where x is the vector of variables that have been fixed. This nogood can serve as a valid constraint throughout the rest of the tree search. Thus at each node one can generate complementary constraints: constraints involving the unfixed variables by means of constraint propagation and cutting plane methods, and nogoods involving the fixed variables.

Again there is cross-fertilization. The optimization community has apparently never used nogoods in branching search. The constraint satisfaction community has apparently never used generalized Benders decomposition

as a means to generate nogoods, although Beringer and de Backer have done related work [46, 18]. The ability of Benders decomposition to exploit structure could give new life to the idea of a nogood, which has received limited attention in practical algorithms.

2.4 DUALITY OF STRENGTHENING AND RELAXATION

A *strengthening* of a problem shrinks the feasible set, and a *relaxation* enlarges it. Solving a strengthened minimization problem provides an upper bound on the optimal value of the original problem. Relaxing the problem provides a lower bound.

The interplay of strengthening and relaxation is an old theme in optimization that goes under the name of *primal-dual* methods. These appear, for example, in dual-ascent and other Lagrangean methods for discrete optimization, out-of-kilter and related methods for network flow problems [12], and the primal-dual simplex method for linear programming. All of these exploit the same formal duality, which is defined below. Branch-and-bound search can be regarded as a primal-dual method in a somewhat different sense that will also be discussed.

Properly chosen strengthenings and relaxations may be much easier to solve than the original problem. Solving several of them and choosing the tightest bounds that result may therefore provide a practical way of bracketing the optimal value of a problem that cannot be solved to optimality.

As noted earlier, enumeration of strengthenings usually takes the form of branching or local search. It is less obvious how to enumerate relaxations of a problem, but a clever method has evolved over the years: one *parameterizes* relaxations. Each parameter setting yields a different relaxation. The problem of finding parameters that yield the tightest bound might be called the *relaxation dual* problem. Several well-known dualities in optimization are special cases, including the linear programming dual, the Lagrangean dual, and the surrogate dual.

These classical duals, however, apply only to problems with inequality constraints. The general relaxation dual may provide a key to relaxing constraints that take other forms. This is particularly important for bringing the advantages of relaxation to constraint satisfaction methods, which permit a much broader repertory of constraints than the inequality constraints of mathematical programming. The first section below suggests how this might be done.

Branch-and-bound search dualizes strengthening and relaxation in a different way. The second section suggests how generalization of this idea can also result in new methods.

2.4.1 Relaxation Duality

We begin by defining strengthening and relaxation more carefully. The definition stated above assumes that the objective function in a strengthening or relaxation is the same as in the original problem. It need not be. The following problem is a *strengthening* of (2.12) in a more general sense if $S' \subset S$ and $f'(x) \geq f(x)$ for $x \in S'$.

$$\begin{array}{ll} \underset{x \in D}{\text{minimize}} & f'(x) \\ \text{subject to} & x \in S' \end{array} \tag{2.21}$$

Problem (2.21) is a *relaxation* of (2.12) if $S' \supset S$ and $f(x) \leq f'(x)$ for $x \in S$. Equivalently, let the *epigraph* E of an optimization problem (2.12) be the set $\{(z, x) \mid z \geq f(x),\ x \in S\}$. A strengthening's epigraph is a subset of E, and a relaxation's epigraph is a superset.

Suppose that a family of relaxations is parameterized by $\lambda \in \Lambda$, so that $f'(x) = f(x, \lambda)$ and $S' = S(\lambda)$. Each relaxation is written

$$\theta(\lambda) = \min_{x \in D} \{f(x, \lambda) \mid x \in S(\lambda)\} \tag{2.22}$$

This is a valid relaxation if:

$$\begin{array}{l} S(\lambda) \supset S, \text{ all } \lambda \in \Lambda \\ f(x, \lambda) \leq f(x), \text{ all } x \in S,\ \lambda \in \Lambda \end{array} \tag{2.23}$$

The problem of finding a relaxation that gives the tightest lower bound is the *relaxation dual*,

$$\begin{array}{ll} \underset{\lambda \in \Lambda}{\text{maximize}} & \theta(\lambda) \end{array} \tag{2.24}$$

The classical *Lagrangean relaxation* replaces the objective function with a lower bound that is obtained by penalizing infeasible solutions and perhaps rewarding feasible ones. It is defined only when the constraints have inequality form, so that $S = \{x \mid g_i(x) \leq 0,\ i \in I\}$. It is obtained by setting $f(x, \lambda) = f(x) + \sum_{i \in I} \lambda_i g_i(x)$ and $S(\lambda) = D$ for $\lambda \geq 0$. Note that the feasible set is the same for all λ. This is a valid relaxation because clearly $S(\lambda) \supset S$, and $f(x) + \sum_{i \in I} \lambda_i g_i(x) \leq f(x)$ for all $\lambda \geq 0$ and all $x \in S$ (so that $g_i(x) \leq 0$). In this case the relaxation dual is the *Lagrangean dual*, which is widely used in integer and nonlinear programming to obtain bounds.

The *surrogate relaxation* leaves the objective function untouched but replaces the inequality constraints with a nonnegative linear combination of those constraints. It is obtained by setting

$$f(x, \lambda) = f(x)$$

and

$$S(\lambda) = \{x \mid \sum_{i \in I} \lambda_i g_i(x) \geq 0\}$$

, with $\lambda \geq 0$. The relaxation dual in this instance is the *surrogate dual*, which can also be used to obtained bounds for integer and nonlinear programming. The Lagrangean and the surrogate dual of a linear programming problem are both equivalent to the linear programming dual.

A relaxation dual can be defined for a much wider range of problems than those involving inequality constraints. It is necessary only that the relaxation observe the formal properties (2.23). This can be illustrated by the traveling salesman problem. The traditional continuous relaxation requires that the problem be written with inequality constraints, resulting in a long problem statement (exponentially long in the most popular formulation). However, the problem can be written very succinctly as follows.

$$\begin{array}{ll} \underset{x \in D}{\text{minimize}} & \sum_i c_{y_i, y_{i+1}} \\ \text{subject to} & \texttt{all-different}\{y_1, \ldots, y_n\} \end{array}$$

where $y_{n+1} = y_1$. Here y_i is the i-th city visited and c_{jk} the cost on arc (j, k). One can of course write a relaxation for this formulation by reverting to the inequality model and relaxing the integrality constraints. But this is not a practical option when an inequality formulation of the problem at hand is unavailable, too large, or has a weak relaxation. A generalized Lagrangean relaxation can perhaps accommodate such cases.

In the case of the traveling salesman problem, a generalized Lagrangean relaxation might be given by

$$f(x, \lambda) = \sum_i c_{y_i, y_{i+1}} + \sum_j \lambda_j (N_j \Leftrightarrow 1) \tag{2.25}$$

and $S(x, \lambda) = \{1, \ldots, n\}$, where N_j is the number of x_i's equal to j [80]. Because (2.25) can be written

$$f(x, \lambda) = \sum_i c_{y_i, y_{i+1}} + \sum_i (\lambda_{y_i} \Leftrightarrow \lambda_i)$$

the value $\theta(\lambda)$ can be readily computed by dynamic programming. The dual (2.24) can be solved by subgradient optimization, because

$$(N_1 \Leftrightarrow 1, \ldots, N_n \Leftrightarrow 1)$$

is a readily available subgradient.

In other types of problems a concept from constraint satisfaction may help provide a useful relaxation of the constraint set. The *dependency graph* G

for a problem (2.12) indicates the extent to which variables decouple [135]. It contains a vertex for each variable and an edge (i, j) when variables x_i and x_j occur in the same constraint or in the same term of the objective function (which we may, for simplicity, assume to be a sum of terms). If vertices (along with adjacent edges) are removed from G in order $1, \ldots, n$, the *induced width* of G with respect to this ordering is the maximum degree of a vertex at the time it is removed.

Problem (2.12) can be solved by nonserial dynamic programming [19] in time that is exponential in the induced width of G. Although the induced width is normally too large for this to be practical, relaxations can be defined for which it is small. This might be done by removing several arcs from G to obtain $G(\lambda)$, where λ is a list of the arcs removed. For each arc (x_i, x_j) removed, replace each constraint containing both x_i and x_j with two projections of the constraint. (The objective function can be analogously treated.) The projections are obtained by projecting the constraint onto all of its variables except x_i and onto all of its variables except x_j. Once the projections are computed, resulting problem has dependency graph $G(\lambda)$.

The relaxed set $S(x, \lambda)$ is now defined to be the projected problem for $G(\lambda)$, and $\theta(\lambda)$ is computed by nonserial dynamic programming. The dual problem (2.23) might be attacked by local search methods over the space of λ's.

These represent only two examples of how discrete relaxations might be parameterized and bounds obtained by solving a relaxation dual. The potential of this approach is largely unexplored.

2.4.2 Searches that Combine Strengthening and Relaxation

The most popular strategy for combining strengthening and relaxation in a search procedure is to enumerate strengthenings and solve a relaxation of each. In integer programming, for instance, one might enumerate strengthenings in a branch-and-bound tree and solve the continuous relaxation of the strengthened problem at each node. A rationale for this strategy is that it hedges against the liabilities of both strengthening and relaxation: strengthenings may not be easy to solve until they become very strong (i.e., almost all variables are fixed), and whereas a continuous relaxation may be easy to solve, it may also be very weak. By solving a relaxation at each node of a search tree, one solves an easy problem that may nonetheless be a relatively strong relaxation because several variables have been fixed. Bounds derived from the relaxations can be used in a branch-and-bound scheme.

This represents only one way that strengthening and relaxation can interact. There are others. For example, the reverse strategy is seldom recognized: one can solve strengthenings of a relaxation. The only requirement is that

the relaxation remain an easy problem when strengthened, for example when variables are fixed. This is normally the case. A feasible solution is found when the solution of a strengthening is feasible in the original problem. One backtracks whenever a feasible solution is found, or it can be determined that no solution of the current strengthening is feasible in the original problem. Bounding can be used as before.

In integer programming, the reverse strategy is identical to the original strategy, because continuous relaxation and variable fixing are commutative functions. Fixing a variable in a continuous relaxation has the same effect as relaxing the problem after fixing that variable. Perhaps this is why the reverse strategy has not been noticed.

In general relaxation and strengthening do not commute. For example, if $x_1, x_2 \geq 0$, the constraint $x_1 x_2 \geq 4$ can be relaxed to $x_1 + x_2 \geq 4$ by writing a first-order Taylor series approximation at the point $(x_1, x_2) = (2, 2)$. The relaxation becomes $x_2 \geq 0$ when x_1 is fixed to, say, 4. Reversing the direction, fixing $x_1 = 4$ changes the nonlinear constraint to $x_2 \geq 1$, which relaxes to itself and is different from $x_2 \geq 0$.

Viewing search consciously as an interplay of strengthening and relaxation can therefore lead one to combine them in different ways and obtain new methods. The effectiveness of these new methods has yet to be tested.

2.5 GENERATING RELAXATIONS VIA INFERENCE

As mentioned earlier, the global constraints of constraint programming provide an opportunity to exploit structure not only for purposes of domain reduction, but for relaxation as well.

To clarify this point, it should be acknowledged that a global constraint is sometimes relaxed in order to compute reduced domains. The result is not, however, normally the sort of relaxation that is recommended here; namely, one that can be solved to optimality in order to obtain useful bounds, such as a linear programming relaxation.

It is true that domain reduction can itself be viewed as a process that generates a relaxation. It in effect derives "in-domain" constraints that restrict each variable to a reduced domain. The constraint programming community normally views in-domain constraints as comprising a *constraint store* that propagates the implications of one constraint to other constraints. But they can also be viewed as comprising a relaxation that is easily solved: merely choose one value from each domain [20]. It may even be practical to optimize the objective function subject to the in-domain constraints. But even in this case, the result is unlikely to provide a useful bound.

The desired sort of relaxation has usually been obtained in the form of cutting planes that are derived from inequality constraints. This imposes a severe limitation, because most useful global constraints are neither expressed nor easily expressible in inequality form. It is often possible, however, to derive linear inequality relaxations, as well as other soluble relaxations, from constraints other than inequalities. This has been done even in traditional operations research for disjunctions of linear inequalities [5, 6, 14]. This and more recent work are summarized in [81].

As an illustration, we present here continuous relaxations for element constraints, which are important due to their role in implementing variable subscripts. To highlight the overall strategy of attaching both domain reduction procedures and relaxations to a global constraint, we also analyze domain reduction for element constraints.

2.5.1 Discrete Variable Subscripts

Variable subscripts are rapidly becoming ubiquitous in modeling of combinatorial optimization problems. The `element` constraint is well known in the constraint programming world [137, 96] as a way of indexing discrete variables, but variable subscripts have now also been introduced in mathematical modeling languages such as AMPL [57, 59] and OPL [138]. While domain reduction ensuring arc- or hyperarc consistency is relatively simple for indexing over discrete variables, the case of continuous variables is much less explored.

The `element` constraint is written,

$$\texttt{element}(y, (v_1, \ldots, v_k), z). \tag{2.26}$$

In the simplest case, y is a single variable whose domain is $\{1, \ldots, k\}$, and $(v_1, \ldots, v_k)$ is a list of values. The variable z can be discrete or continuous. The constraint says that z must take the y-th value in the list.

An `element` constraint of this form implements a term with a variable subscript. A term of the form $c_{f(y)}$, where $f(y)$ is a function of the variable y, is implemented by imposing the constraint

$$\texttt{element}(y, (c_{f(1)}, \ldots, c_{f(k)}), z)$$

and replacing all occurrences of $c_{f(y)}$ with z. For example, if $y \in \{1, 2, 3, 4\}$ the term $c_{y,y+1}$ is replaced by z and the constraint

$$\texttt{element}(y, (c_{12}, c_{23}, c_{34}, c_{45}), z).$$

Because the simplest `element` constraint (2.26) contains only two variables, arc consistency is equivalent to full consistency. It is achieved in the obvious

way. Let D_z, D_y be the current domains of z and y, respectively, and let $\bar{D}_z, \bar{D}_y$ be the new, reduced domains. Then the two rules, $\bar{D}_z = D_z \cap \{v_j \mid j \in D_y\}$ and $\bar{D}_y = D_y \cap \{j \mid v_j \in D_z\}$, in a fix-point iteration, will achieve arc consistency.

Indexing among values is just an instance of the general case of variables (as opposed to constants) with variable subscripts. Because `element` now contains $k+2$ variables, arc consistency does not imply hyperarc consistency. However, full hyperarc consistency can be obtained as follows (of which the rules above are a special case).

(a) The domain of z must be a subset of the combined domains of the variables x_j for which j belongs to the domain of y. So

$$\bar{D}_z = D_z \cap \bigcup_{j \in D_y} D_{x_j}.$$

(b) The domain of y is restricted to indices j for which the domain of z intersects the domain of x_j. Thus

$$\bar{D}_y = D_y \cap \{j \mid D_x \cap D_{x_j} \neq \emptyset\}.$$

(c) The domain of x_j can be restricted if j is the only index in the new domain $\bar{D}_y$ of y.

$$\bar{D}_{x_j} = \begin{cases} \bar{D}_z, & \text{if } \bar{D}_y = \{j\}, \\ D_{x_j}, & \text{otherwise.} \end{cases}$$

Example 1 Consider the `element` constraint

$$\texttt{element}(y, (x_1, x_2, x_3, x_4), z)$$

where initially the domains are:

$$\begin{aligned} D_z &= \{20, 30, 60, 80, 90\} \\ D_y &= \{1, 3, 4\} \\ D_{x_1} &= \{10, 50\} \\ D_{x_2} &= \{10, 20\} \\ D_{x_3} &= \{40, 50, 80, 90\} \\ D_{x_4} &= \{40, 50, 70\} \end{aligned}$$

Rules (a), (b) and (c) imply that the reduced domains are:

$$\begin{aligned} \bar{D}_z &= \{20, 30, 60, 80, 90\} \cap \{10, 40, 50, 70, 80, 90\} = \{80, 90\} \\ \bar{D}_y &= \{1, 3, 4\} \cap \{3\} = \{3\} \\ \bar{D}_{x_1} &= D_{x_1} \\ \bar{D}_{x_2} &= D_{x_2} \\ \bar{D}_{x_3} &= \bar{D}_z = \{80, 90\} \\ \bar{D}_{x_4} &= D_{x_4} \end{aligned}$$

Thus y is fixed to 3, which means that $x_3 = z$. The common domain of x_3 and z is the intersection of their original domains. □

2.5.2 Continuous Variable Subscripts

While the discrete cases above are well known, variable subscripts in continuous linear inequalities are far less explored. In this case, constraint propagation is replaced by cutting plane generation. Suppose for example that x is a continuous variable in the constraint $x_{f(y)} \geq \beta$. The inequality $z \geq \beta$ is inserted into the LP model along with additional constraints that define z. If the value of y is fixed, e.g, to 1, one adds the constraint $z = x_{f(1)}$. If the current domain of y is $\{1, 2\}$, however, the constraint that defines z is a disjunction:

$$(x_{f(1)} = z) \vee (x_{f(2)} = z). \tag{2.27}$$

Although (2.27) cannot be added to a linear model, a linear relaxation of it, defined by cutting planes, can be used instead.

In general a subscripted variable $x_{f(y)}$ is represented by replacing it with z and the linear relaxation of the general disjunction

$$\bigvee_{i \in D_y} x_{f(i)} = z. \tag{2.28}$$

In order to be useful, the variables x_j must also have bounds, such as $0 \leq x_{f(j)} \leq m_{f(j)}$ for $j \in D_y$. We assume in what follows that $|D_y| \geq 2$.

If each upper bound is the same value m_0, we get the following valid inequalities for (2.28):

$$\sum_{i \in D_y} x_{f(i)} \leq z + (|D_y| \Leftrightarrow 1) m_0, \tag{2.29}$$

$$\sum_{i \in D_y} x_{f(i)} \geq z, \tag{2.30}$$

$$0 \leq x_{f(i)} \leq m_0, \quad i \in D_y, \tag{2.31}$$

$$0 \leq z \leq m_0. \tag{2.32}$$

The inequality (2.30) is a surrogate inequality [6] and the bounds (2.31)–(2.32) are from before.

Let $\mathcal{C}$ be the polyhedron defined by (2.28) and (2.31)–(2.32), and let $\mathcal{P}$ be the polyhedron defined by (2.29)–(2.32) It can be easily shown that $\mathcal{C} \subseteq \mathcal{P}$, since points in $\mathcal{C}$ are convex combinations of points where at least one of the x_i's is equal to z. Equations (2.29)–(2.30) follow immediately.

We note that the inequalities defining $\mathcal{P}$ are facets of the convex hull relaxation of $\mathcal{C}$. The $|D_y|+1$ points, $(0, m_0, \ldots, m_0, 0)$, $\ldots$, $(m_0, \ldots, m_0, 0, 0)$ and $(m_0, \ldots, m_0)$ in $\mathcal{C}$ are affinely independent and satisfy (2.29) at equality. Also, the $|D_y|+1$ points $(m_0, 0, \ldots, 0, m_0)$, $\ldots$, $(0, \ldots, 0, m_0, m_0)$ and $(0, \ldots, 0)$ in $\mathcal{C}$ are affinely independent and satisfy (2.30) at equality. The bounds (2.31)–(2.32) are obviously facets of $\mathcal{C}$. It is shown in [78] that $\mathcal{P}$ is in fact the convex hull of the disjunction (2.28).

If the upper bounds differ the convex hull relaxation can be much more complex. In this case one can use a weaker and simpler relaxation by letting $m_0 = \max_i\{m_{f(i)}\}$ in (2.29) and (2.32) and replacing (2.31) with the actual bounds.

One can augment this relaxation with second relaxation. First write the disjunction (2.28) in the weaker form of two disjunctions,

$$\bigvee_{i \in D_y} (x_{f(i)} \Leftrightarrow z \geq 0), \tag{2.33}$$

$$\bigvee_{i \in D_y} (\Leftrightarrow x_{f(i)} + z \geq 0), \tag{2.34}$$

and replace each with the linear "elementary" relaxation described in [14]. This yields,

$$\sum_{i \in D_y} \frac{x_i}{m_{f(i)}} \Leftrightarrow \left(\sum_{i \in D_y} \frac{1}{m_{f(i)}} \right) z \geq \Leftrightarrow |D_y| + 1, \tag{2.35}$$

$$\Leftrightarrow \sum_{i \in D_y} \frac{x_i}{m_{f(i)}} + \left(\sum_{i \in D_y} \frac{1}{m_{f(i)}} \right) z \geq \Leftrightarrow |D_y| + 1, \tag{2.36}$$

When all upper bounds are the same, $m_0 = \max_i\{m_{f(i)}\}$, (2.35)–(2.36) become the following, which are dominated by (2.29)–(2.30):

$$\sum_{i \in D_y} x_{f(i)} \geq |D_y| z \Leftrightarrow (|D_y| \Leftrightarrow 1) m_0, \tag{2.37}$$

$$\sum_{i \in D_y} x_{f(i)} \leq |D_y| z + (|D_y| \Leftrightarrow 1) m_0. \tag{2.38}$$

But when the upper bounds differ, it is advantageous to use both (2.29)–(2.30) and (2.35)–(2.36) along with the upper bounds and (2.32), where $m_0 = \max_i\{m_{f(i)}\}$ in (2.29)–(2.30).

Example 2 The goal is to generate linear inequalities to represent $x_y \geq \beta$ in a linear programming solver, where the current domain of y is $\{1, 2\}$.

Suppose initially that $0 \leq x_j \leq 5$ for $j = 1, 2$. Then one can generate the inequality $z \geq \beta$ and define z with the relaxation of (2.27). The latter is given by (2.29)–(2.32), which in this case is

$$0 \leq x_1 + x_2 \Leftrightarrow z \leq 5, \tag{2.39}$$
$$0 \leq x_1, x_2, z \leq 5, \tag{2.40}$$
$$0 \leq z \leq 5, \tag{2.41}$$

Thus the constraints $z \geq \beta$ and (2.39)–(2.41) are added to the LP model.

Now suppose the upper bounds are different: $0 \leq x_1 \leq 4, 0 \leq x_2 \leq 5$. The elementary relaxation in (2.35)–(2.36) becomes:

$$5x_1 + 4x_2 \Leftrightarrow 9z \leq 20,$$
$$5x_1 + 4x_2 \Leftrightarrow 9z \geq \Leftrightarrow 20.$$

These inequalities are combined with $z \geq \beta$, (2.39), (2.41) and the bounds $0 \leq x_1 \leq 4, 0 \leq x_2 \leq 5$ in the LP model. □

An alternative to the disjunctive relaxation discussed above would be to use a variant of the conventional "big-M" formulation. Equations (2.33)–(2.34) would then form a relaxation of (2.28) as

$$x_{f(i)} \Leftrightarrow z \geq \Leftrightarrow M(1 \Leftrightarrow y_i), \qquad i \in D_y, \tag{2.42}$$
$$\Leftrightarrow x_{f(i)} + z \geq \Leftrightarrow M(1 \Leftrightarrow y_i), \qquad i \in D_y, \tag{2.43}$$
$$\sum_{i \in D_y} y_i = 1, \tag{2.44}$$
$$0 \leq y_i \leq 1, \qquad \forall i \in D_y, \tag{2.45}$$

where $M = \max_i\{m_{f(i)}\}$. Note that we introduce $|D_y|$ new continuous variables in this relaxation. The projection of (2.42)–(2.45) on variables $(x_{f(1)}, \ldots, x_{f(|D_y|)}, z)$ is very weak so as a relaxation the big-M formulation is not only more costly ($|D_y|$ new variables and $2|D_y| + 1$ new constraints) but is almost useless. Its use is in a search framework where y is needed for branching purposes, i.e., where the framework does not allow branching on constraints, e.g., on parts of a disjunction. Instead, the y's in the big-M formulation above are used to simulate that capability.

2.5.3 Incremental Cutting Plane Generation

Although useful on their own, the relaxations for variable subscripts described in the previous section are mainly intended for use within a branch-and-bound search. Due to branching and inference (such as constraint propagation), the domain of the indexing variable y will shrink as we descend in

the search tree. This means that $x_{f(i)}$ should be removed from the equations when $i \notin D_y$ (by setting the corresponding coefficient to zero), and also that the coefficients $|D_y|$ and m_j need to be updated in equations (2.29)–(2.38) when D_y is modified.

This last comment is related to how the M's in a conventional "big-M" formulation are selected and handled. Ideally, they should be updated when the variable bounds change, to obtain the strongest possible relaxation, but often this seems to be neglected.

2.6 FUTURE RESEARCH DIRECTIONS

A number of research directions are identified in the foregoing. They may be summarized as follows.

- *Deciding what to relax.* Learn how to identify subsets of constraints that have a useful continuous relaxation.
- *Continuous relaxations for global constraints.* Find useful continuous relaxations for common global constraints.
- *Relaxation duals.* Use the idea of a relaxation dual to create discrete relaxations for common global constraints.
- *Sensitivity analysis.* Develop inference-based sensitivity analysis for problem classes by analyzing when problem perturbations leave the proof of optimality (or infeasibility) intact. Also, learn how to generalize this analysis so as to explain the solution.
- *Using nogoods in branch-and-cut search.* Investigate the possibility of using nogoods obtained by inference-based Benders decomposition as cuts that are complementary to the traditional cuts in branch-and-cut search.
- *Finding nogoods that exploit structure.* Use generalized Benders decomposition as a means to identify nogoods that exploit problem structure and perhaps thereby improve the utility of nogoods.
- *Strengthening and relaxation.* Experiment with new ways for combining strengthening and relaxation during search, for instance by solving strengthenings of a relaxation.
- *Unified solution technology.* Solve a wide variety of problems with a view to how the search/inference and strengthening/relaxation dualities may be exploited, with the aim of building a solution technology that unifies and goes beyond classical optimization and constraint satisfaction methods.

A Declarative Modeling Framework that Integrates Solution Methods

John N. Hooker, Hak-Jin Kim and Greger Ottosson

Constraint programming offers modeling features and solution methods that are unavailable in mathematical programming but are often flexible and efficient for scheduling and other combinatorial problems. Yet mathematical programming is well suited to declarative modeling languages and is more efficient for some important problem classes. This raises this issue as to whether the two approaches can be combined in a declarative modeling framework. This paper proposes a general declarative modeling system in which the conditional structure of the constraints shows how to integrate any "checker" and any special-purpose "solver." In particular this integrates constraint programming and optimization methods, because the checker can consist of constraint propagation methods, and the solver can be a linear or nonlinear programming routine.

3.1 INTRODUCTION

Solution technology has strongly influenced the modeling framework of mathematical programming. Inequality constraints, for example, are ubiquitous not only because they are useful for problem formulation, but because solvers can deal with them. Linear programming is tethered to a highly structured modeling language because the solver requires it. Its historical

popularity has been possible only because of George Dantzig's discovery that its restricted vocabulary is surprisingly versatile in applications.

Nonetheless mathematical programmers are becoming more aware of the limitations of inequality-based modeling. This may be due primarily to the recent commercial success of constraint programming and its richer modeling resources. Constraint programming systems permit not only such logical constructions as disjunctions and implications [137, 140, 96], but they include high-level symbolic constraints – such as all-different [123, 137] and scheduling constraints [1, 33] – and other highly useful predicates that are foreign to a mathematical programming environment [1, 16, 34, 140]. Variables need not even have numerical values, and they can appear in subscripts. The power of this modeling framework is dramatically illustrated by the traveling salesman problem, which requires exponentially many constraints in its most popular integer programming formulation. In a constraint programming milieu it can be written with a single all-different constraint, if variable subscripts are permitted in the objective function.

Two things about modeling in mathematical programming, however, are arguably very right. One is that modeling takes place within a fully declarative modeling language, such as AMPL [58, 59] or GAMS [24, 25]. The modeler can state the model without describing the solution procedure, and the model can be passed to any number of solvers without modification. Constraint programming systems, by contrast, traditionally lack a modeling language front end. Rather, they are integrated in some general programming language (such as C++ or Prolog) and may therefore be less accessible and harder to use. Customized problem-specific search procedures are common and often necessary. A recent effort made to overcome the gap between mathematical modeling languages and constraint programming systems is OPL [138], which allows mixing of discrete and continuous constraints and provides a high-level way of describe problem-specific search strategies.

A second strength of mathematical programming is that the very structure of the modeling language anticipates the solution task. Although the human modeler may think little about algorithmic matters, the syntax of the language forces the model into the solver's mold.

Although mathematical programming enjoys these advantages, it would benefit from the larger modeling repertory of constraint programming and the solution techniques that go with it. Fourer [57] proposed an extension of AMPL that incorporates modeling devices from constraint programming, but it does not address the issue of how solvers might cooperate. The issue is whether there is a principled way for a modeling system to integrate optimization and constraint satisfaction methods, without obliging the modeler to think about how the corresponding solution methods will interact. More

generally, is there a general way in which a modeling language can be sensitive to available solution technology while remaining declarative? This paper is intended to address these questions.

The Mixed Logic/Linear Programming (MLLP) framework developed in [81] addresses them in a limited way, and it serves as the starting point for the present study. MLLP assumes that a linear programming solver and a branching mechanism are available. Its constraints are written as conditional statements of the form $D \rightarrow C$, where the antecedent D is a constraint involving discrete variables, and the consequent C a system of linear inequalities. This conditional structure relates to a branching algorithm in a natural way. At each node of the branching tree, the values of the discrete variables may be fixed or restricted in such a way as to satisfy some of the antecedents D. The corresponding consequents C form the constraint set of a linear optimization problem that is passed to the linear programming solver. Checking whether partially determined discrete variables satisfy the antecedents is an inference problem that can be attacked with constraint propagation and domain reduction methods developed in the constraint satisfaction community. The MLLP framework therefore unites linear programming and constraint satisfaction in a way that is dictated by the structure of the constraints. In addition, the language architecture not only provides the useful modeling devices associated with constraint satisfaction but captures continuous and discrete elements in a way that appears convenient for a wide range of problems.

Whereas MLLP was developed in an intuitive fashion, the intent here is to understand more self-consciously how it unites different solution methods, in order to develop a more general approach. We first observe that it integrates a *checker* with a *solver*. The checker consists of finite-domain constraint satisfaction techniques that determine when antecedents are satisfied. The solver is a linear programming algorithm that is applied to the consequents. The same style of integration can be used for other checkers and solvers. For example, if one branches on continuous as well as on discrete variables, the checker could take the form of interval arithmetic or other techniques for maintaining bounds consistency. The solver could be a nonlinear programming algorithm or an efficient algorithm for solving special classes of discrete problems. Thus if we can specify in general how the conditional constraints interact with a branching mechanism, a template for uniting a wide range of solution techniques is available.

This paper first reviews MLLP with the benefit of hindsight so as to point out features that will emerge in the more general treatment to follow. It proceeds to develop an abstract modeling framework that links conditional constraints with a generic branching algorithm. We discover that the branching strategy so often used in optimization and constraint satisfaction has a more complex structure than one might imagine. We attempt to delineate this

structure precisely so that its connection with the model can be made precise. This permits us to state and prove sufficient conditions under which the combination of solution methods specified by a given model will in fact yield a solution.

Our purpose is not to propose a new algorithm nor even to suggest exactly how existing algorithms should be linked. Rather, we show how the syntactic structure of a declarative modeling language can specify a linkage. We take it for granted that it is often worthwhile to combine optimization and constraint satisfaction methods. This appears now to be widely accepted in the two communities and in any case is well documented in [81] and the references cited there.

3.2 MIXED LOGICAL/LINEAR PROGRAMMING

Mixed logical/linear programming (MLLP) solves problems by branching on discrete variables. It assumes the availability of constraint satisfaction methods for checking whether a partial solution satisfies discrete constraints, and a linear programming (LP) solver for continuous constraints. In principle the discrete constraints may take any form, provided it is possible to check whether an assignment of values to all of the discrete variables satisfies them. Some restriction on the form of constraints may be necessary, however, if constraint propagation methods are to help determine whether a partial assignment of values satisfies the constraints. The continuous constraints take the form of systems of linear inequalities.

As noted earlier, the constraints are conditionals in which the antecedents are discrete and the consequents continuous. Discrete variables may also appear in the consequents in subscripts and index sets over which sums, etc., are taken. Thus when the solution algorithm branches on values of the discrete variables, some of the antecedents become true at a given node of the search tree. The inequalities in the corresponding consequents are thereby enforced and must be simultaneously satisfied. A feasible solution is obtained when the truth value of every antecedent is determined, and the LP solver finds a solution for the enforced inequalities. Computational tests reported in [81] suggest that an MLLP framework not only has modeling advantages but can often permit more rapidly solution of the problem than traditional mixed integer programming solvers.

The modeling language is first described in detail below and some examples given. The solution algorithm is then presented. Subsequently MLLP is contrasted with traditional mixed integer programming.

3.2.1 The Modeling Language

An MLLP model in its most basic form is,

$$\begin{array}{ll} \min & cx \\ \text{s.t.} & h_i(y) \rightarrow A^i x \geq b^i, \quad i \in I \\ & x \in R^n, y \in D. \end{array} \tag{3.1}$$

where y is a vector of discrete variables and x a vector of continuous variables. $h_i(y)$ is a constraint that contain only variables in y. It is regarded as true when y satisfies it.

A linear constraint set $Ax \geq b$ that is enforced unconditionally can be formally written as a conditional constraint,

$$(0 = 0) \rightarrow Ax \geq b.$$

Similarly, an unconditional discrete constraint h can be formally represented with the conditional $\neg h \rightarrow (1 = 0)$. In the following, however, no pains will be taken to put constraints in conditional form when they hold categorically.

One of the more useful modeling constructs in practice is a disjunction of linear systems, which is easily written in conditional form. For example, a fixed charge cost function

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ F + cx & \text{if } x > 0 \end{cases}$$

is naturally expressed by replacing $f(x)$ with the continuous variable z and adding the disjunctive constraint,

$$(z = x = 0) \vee \begin{pmatrix} x > 0 \\ z = F + cx \end{pmatrix}.$$

The disjunction can be written in conditional form as,

$$\begin{array}{l} (y = 0) \rightarrow (x = z = 0) \\ (y = 1) \rightarrow (x > 0, z = F + cx), \end{array}$$

where y has the domain $\{0, 1\}$.

Another useful modeling device is a subscript that contains a discrete variable. Suppose for example that in an assignment problem, the cost of assigning worker k to job j is c_{jk}. Then the objective function can be written $\sum_j c_{jy_j}$, where y_j is a discrete variable that indicates the worker assigned job j. The value of c_{jy_j} is in effect a function of $y = (y_1, \ldots .Y_n)$ and can be written $g_j(y)$, where function g_j happens to depend only on y_j. $g_j(y)$ might be called a *variable constant*, despite the oxymoron. Similarly, a continuous

variable's subscript can be a function of y, resulting in a *variable subscript*. The MLLP model can incorporate these devices as follows.

$$\begin{array}{ll} \min & cx \\ \text{s.t.} & h_i(y) \rightarrow L_i(x,y), \quad i \in I \\ & x \in R^n, y \in D, \end{array} \tag{3.2}$$

where

$$L_i(x,y) = \sum_{k \in K_i(y)} a_{ik}(y) x_{j_{ik}(y)} \geq b_i(y).$$

Here the coefficient $a_{ik}(y)$ and right-hand side $b_i(y)$ are variable constants. The subscript $j_{ik}(y)$ is a variable subscript. In addition the summation is taken over a *variable index set* $K_i(y)$.

Models of this sort can in principle be written in the more primitive form (3.1) by adding sufficiently many conditional constraints. For example, the constraint $z \geq \sum_j c_{jy_j}$ can be written $z \geq \sum_j z_j$, if the following constraints are added to the model,

$$(y_j = k) \rightarrow (z_j = c_{jk}), \quad \text{all } j,k \in \{1, \ldots n\},$$

where each $y_j \in \{1, \ldots, n\}$. It is inefficient to unpack (3.2) in this manner, however. The solution algorithm will deal directly with variable constants, subscripts and index sets simply by enforcing a constraint that contains them only after enough y_j's are fixed to determine their values.

Because the conditional structure is related to the solution method, it can distinguish hard constraints that must be attacked with branching from easier constraints that can be passed to the solver. For example, it is obviously easy to check satisfiability of the constraint

$$(y = 1) \rightarrow (Ax \geq b),$$

because it requires only one branch. One first checks whether $y = 1$ and, if so, uses a linear solver to determine whether $Ax \geq b$ is satisfiable. The language excludes, however, the converse expression,

$$(Ax \geq b) \rightarrow (y = 1).$$

It must be written in some fashion that reveals the hidden disjunction in the constraint, such as,

$$\begin{array}{l} (y' = 1) \vee \ldots \vee (y' = m) \vee (y = 1) \\ (y' = i) \rightarrow (A^i x < b_i), \quad i = 1, \ldots, m, \end{array}$$

where $A^1, \ldots, A^m$ are the rows of A. The necessity of enumerating several possibilities is now evident.

3.2.2 Examples

The modeling language developed above is already considerably more expressive than that of traditional mathematical programming. This is illustrated by the following examples.

- *Traveling salesman problem.* It can be compactly written,

$$\begin{array}{ll} \min & z \\ \text{s.t.} & z \geq \sum_{j=1}^{n} c_{y_j y_{j+1}}, \\ & \text{all-different}\{y_1, \ldots, y_n\} \end{array}$$

  where y_j has domain $\{1, \ldots, n\}$, and y_{n+1} is identified with y_1. Here y_j is the j-th city visited by the salesman. A slightly more sophisticated model is,

$$\begin{array}{ll} \min & z \\ \text{s.t.} & z \geq \sum_{j=1}^{n} c_{j y_j}, \\ & \text{cycle}\{y_1, \ldots, y_n\}, \end{array}$$

  where y_j is now the city that is visited after city j. The cycle constraint says simply that the y_j's define a tour [34]. That is, the cities visited $q_1, \ldots, q_n$ are all different, where the q_j's are defined by $q_1 = 1$ and $q_{j+1} = y_{q_j}$ for each $j < n$.

- *Quadratic assignment problem.* It also has an elegant formulation. Suppose that each facility j is assigned to a distinct location y_j, so that for any pair of facilities j, k, a cost $c_{jk} d_{y_j y_k}$ is incurred. The problem can be written,

$$\begin{array}{ll} \min & z \\ \text{s.t.} & z \geq \sum_{j,k} c_{jk} d_{y_j y_k} \\ & \text{all-different}\{y_1, \ldots, y_n\} \end{array}$$

- *Piecewise linear functions.* A continuous piecewise linear objective function $f(x)$ given by

$$f(x) = f_k + a_k(x \Leftrightarrow v_k) \text{ if } v_k \leq x \leq v_{k+1}, \text{ for } k = 1, \ldots, n$$

  can be written $f_y + a_y(x \Leftrightarrow v_y)$ if one adds the constraint

$$v_y \leq x \leq v_{y+1},$$

  where $y \in \{1, \ldots, n\}$ indicates which segment $[v_k, v_{k+1}]$ contains x.

- *One-machine scheduling problem.* A model that minimizes tardiness can be written,

$$\begin{aligned} \min \quad & \sum_j z_j \\ \text{s.t.} \quad & z_j \geq t_j + d_{y_j} \Leftrightarrow f_{y_j}, \quad \text{all } j \\ & t_{j+1} \geq t_j + d_{y_j} + s_{y_j y_{j+1}}, \quad \text{all } j < n \\ & \text{all-different}\{y_1, \ldots, y_n\} \\ & t_1 = 0, \end{aligned}$$

  where t_j is the time at which the j-th scheduled job begins, d_i is the duration of job i, $s_{ii'}$ is the setup time for job i' when preceded by job i, and f_i is the due date for job i.

- *One-machine problems with resource constraint.* A variation of the one-machine problem that is difficult to model in integer programming imposes a resource capacity constraint. Rather than require the jobs to run sequentially, it allows jobs to be processed simultaneously as long as their total resource consumption rate does not exceed the limit. If each job i consumes resources at the rate r_i, then the sum of r_i over all jobs i in process at any given moment must be no greater than R. The problem is easily formulated as follows if t_i is the start time of job i, and variable index sets are used. The key is that resource consumption need only be checked at times when jobs begin processing.

$$\begin{aligned} \min \quad & \sum_i z_i \\ \text{s.t.} \quad & z_i \geq t_i + d_i \Leftrightarrow f_i, \quad \text{all } j \\ & \sum_{k \atop t_k \leq t_i \leq t_k + d_k} r_k \quad \leq R, \quad \text{all } i \\ & t_i \geq 0, \quad \text{all } i. \end{aligned}$$

  Here z_i measures the tardiness of job i. Unfortunately this model is not in the proper form (3.2), because the variable index set $\{k \mid t_k \leq t_i \leq t_k + d_k\}$ is a function of continuous rather than discrete variables. The LP portion of the problem cannot be solved until the contents of the index sets are defined, which in turn cannot occur until the LP is solved in order to fix the continuous variables.

  Defining a series of *events*, $e \in \{1, \ldots, 2n\}$, allows us to escape this dilemma. U_e is the type of event e, with $U_e = S$ when e is the start of a job and $U_e = F$ when it is the finish. t_e is the time of event e and j_e is the job involved. The resources being consumed at the time of

event e are those consumed by all jobs that started at or before event e, minus those consumed by all jobs that finished before event e.

$$\begin{aligned}
\min \quad & \sum_e z_e \\
\text{s.t.} \quad & (U_e = F) \rightarrow (z_e \geq t_e \Leftrightarrow f_{j_e}), \quad \text{all } e \\
& (U_e = S) \rightarrow \left(z_e = 0, \quad \sum_{\substack{e' \leq e \\ U_{e'} = S}} r_{j_{e'}} \Leftrightarrow \sum_{\substack{e' < e \\ U_{e'} = F}} r_{j_{e'}} \leq R \right), \quad \text{all } e \\
& (j_e = j_{e'}) \rightarrow (U_e = S, U_{e'} = F, t_{e'} = t_e + d_{j_e}), \text{all } e, e' : e < e'. \\
& t_1 = 0, \; t_e \geq t_{e-1}, \quad \text{all } e.
\end{aligned}$$

Here z_e is the tardiness of the job whose completion is event e, if e is a completion event; otherwise $z_e = 0$. The second constraint checks each start event and makes sure that the resources being consumed at that time total less than R.

The third constraint must be imposed for every pair of events e, e' with $e < e'$. The constraint can be written more compactly as

$$t_{F_j} = t_{S_j} + d_j, \quad \text{all } j$$

if the following constraints are also added.

$$j_{S_k} = j_{F_k} = k, \quad \text{all } k. \tag{3.3}$$

Here S_k indicates which event is the start of job k's processing, and F_k indicates which event is the finish. The constraints (3.3) in a sense dualize the indices.

The above approach essentially sees the problem as a discrete event simulation and checks for constraint satisfaction at discrete events. A similar principle can accommodate a wide variety of problems that are difficult to capture in mixed integer programming.

3.2.3 The Solution Algorithm

Solution of an MLLP model proceeds by enumerating partial assignments to y. For each, it checks whether y satisfies the antecedents of the constraints and solves an LP subproblem consisting of inequalities in the enforced consequents.

More precisely, the algorithm conducts a tree search by branching on values of the discrete variables y_j. When branching on y_j, the algorithm defines each branch by restricting the domain of y_j to a different subset $\bar{D}_j$ of the

original domain D_j. y_j is fixed to a particular value when D_j is restricted to a singleton.

Each node of the tree is processed in two steps. First, a *checker* determines, for each antecedent $h_i(y)$, whether the restricted domains $\bar{D}_j$ satisfy it, violate it, or neither. The domains satisfy $h_i(y)$ when any $y \in \bar{D}_1 \times \ldots \times \bar{D}_n$ satisfies $h_i(y)$, and similarly for violation. Suppose for example that $\bar{D}_1 = \{1, 2\}$ and $\bar{D}_2 = \{1, 2, 3\}$. These domains satisfy $y_1 + y_2 \geq 2$, violate $y_1 + y_2 \geq 6$, and neither satisfy nor violate $y_1 + y_2 \geq 4$. A constraint is *determined* by the domains if they satisfy or violate it. The checker must be *correct*, in that it does not judge a constraint to be satisfied or violated unless it really is. It may, however, be *incomplete*, meaning that it may fail to recognize that a constraint is satisfied or violated. When all the y_j's have been fixed, however, the checker should be complete.

The second step is to use the LP *solver* to minimize the objective function cx subject to a set L of linear constraints, defined as follows. We will say the constraint set $L_i(x, y)$ is *enforced* if the checker finds y to satisfy $h_i(y)$. It is *determined* if the values of all variable constants, subscripts and index sets in $L_i(x, y)$ are determined by the domains. Now L is the union of all enforced constraint sets $L_i(x, y)$ that are determined by the domains $\bar{D}_j$.

A feasible solution is discovered when each $h_i(y)$ and each enforced constraint set $L_i(x, y)$ is determined, and the solver obtains a feasible solution. In this event, the y_j's may be assigned any value in their domains, and x may be assigned the optimal solution obtained by the LP solver, in order to obtain a feasible solution (x, y). The choice of y does not affect the value of the objective function because y does not occur there.

When a feasible solution is discovered, or the solver finds no feasible solution for the LP subproblem, the search backtracks. Otherwise it continues to branch. The precise algorithm appears in Fig. 3.1.

The checker can employ a wide gamut of techniques to determine whether the current domains satisfy or violate the constraints $h_i(y)$. In particular it can use constraint satisfaction methods, many of which are designed to reduce the size of domains by constraint propagation and other techniques. The checker therefore provides a general way of understanding how constraint satisfaction and optimization methods might be integrated.

3.2.4 Comparison with Integer Programming

It is instructive to compare MLLP with traditional integer and mixed integer linear programming (MILP). MILP seems similar in that it, too, solves an LP subproblem at each node of a branch-and-bound tree. But this resemblance can be misleading. In MLLP, the LP subproblem contains only the original continuous variables, whereas in MILP it contains all the

```
Let A be a set of active nodes,
   each defined by a domain vector D̄ = (D̄_1, ..., D̄_n),
   where D̄_j is the current domain of y_j.
Initially A = {D}, where the D = (D_1, ..., D_n) is the
original domain vector.
Let z̄ be an upper bound on the optimal value, initially ∞.
While A is nonempty:
   Remove a node D̄ from A.
   Let L be the union of all constraint sets L_i(x, y) that
      are enforced and determined by D̄.
   Let z be the minimum value of the LP subproblem of
   minimizing cx subject to L.
   If z < ∞ (i.e., L is feasible) then
      If each h_i(y) and each enforced L_i(x, y) is determined then
         If z < z̄ then
            Let z̄ = z.
            If z̄ > −∞ then let x̄ be an optimal solution of the
               LP subproblem and ȳ any element of D̄_1 × ... × D̄_n.
      Else branch by adding nodes D̄^1, ..., D̄^K to A,
      where D̄^k_j ⊂ D̄_j for each j, k.
If z̄ = ∞ then the problem is infeasible.
Else if z̄ > −∞ then (x̄, ȳ) is an optimal solution.
Else the problem is unbounded.
```

Figure 3.1: A generic branch-and-bound algorithm for MLLP.

variables (with the integrality constraints relaxed). In fact, if one were to solve a pure integer programming problem in an MLLP framework, the LP subproblems would be empty.

If this seems odd, it is perhaps because traditional MILP has not clearly distinguished the LP subproblem from an LP *relaxation*. Given a problem of the form,

$$\begin{aligned} \min \quad & cx + dy \\ \text{s.t.} \quad & Ax + By \geq b \\ & y_j \text{ integer, all } j, \end{aligned} \tag{3.4}$$

MILP typically solves the following relaxation at each node:

$$\begin{aligned} \min \quad & cx + dy \\ \text{s.t.} \quad & Ax + By \geq b \\ & u \leq y \leq v, \end{aligned} \tag{3.5}$$

or perhaps some enhancement such as Lagrangean relaxation. Here u, v are bounds that are imposed by branching on integer variables.

The most straightforward way to write (3.4) in an MLLP context is the following.

$$\begin{aligned} \min \quad & cx + dy \\ \text{s.t.} \quad & Ax + By \geq b \\ & (\bar{y}_j = k) \rightarrow (y_j = k), \text{ all } j \\ & \bar{y}_j \in \{0, \ldots, K\}, \text{ all } j, \end{aligned} \tag{3.6}$$

where it is assumed that each $y_j \leq K$. The discrete variables $\bar{y}_j$ have continuous counterparts y_j in the linear part of the problem. By branching on the $\bar{y}_j$'s one in effect branches on integer values of the y_j's. The linear relaxation of the problem is solved at each node of the search tree and provides bounds as it does traditionally. Similar schemes have been proposed by Bockmayer [21] as well as Rodošek *et al.* [127], who combine ECLiPSe and CPLEX.

A more idiomatic rendering for MLLP, however, is the following.

$$\begin{aligned} \min \quad & cx + z && (3.7) \\ \text{s.t.} \quad & z \geq dy && (3.8) \\ & Ax \geq b \Leftrightarrow By && (3.9) \\ & z \geq 0 && (3.10) \\ & y_j \in \{0, \ldots, K\}, \text{ all } j, \end{aligned}$$

where dy and By are interpreted as variable constants. The LP subproblem at a node minimizes (3.7) subject to (3.10) alone until branching begins to fix y_j's. Each row of (3.9) enters the LP when the y_j's occurring in that row are fixed, and similarly for (3.8).

The option of solving relaxations, including the relaxation (3.5), is available to MLLP as well. Formally, if the LP subproblem is to minimize cx subject to L, MLLP can solve the relaxation

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & L' \cup \{y \in \bar{D}_1 \times \ldots \times \bar{D}_n\}, \end{aligned} \tag{3.11}$$

where $\bar{D}_1, \ldots, \bar{D}_n$ are the current domains. Here L' is a linear constraint set that need only provide a valid lower bound; i.e., the minimum value of (3.11) is a lower bound on the minimum value of (3.7)-(3.10) when the constraint $y \in \bar{D}_1 \times \ldots \times \bar{D}_n$ is added to the latter. In the conventional relaxation (3.5), each D_j is described by $u_j \leq y_j \leq v_j$, and L' consists of the entire constraint set $Ax + By \geq b$. Unlike traditional MILP, however, MLLP has the option of solving relaxations that do not contain the integer variables. As documented in [79, 81], this can be useful when the traditional relaxation is weak or can be replaced by a more succinct relaxation.

3.2.5 Integrating Other Checkers and Solvers

It was noted earlier that the conditional constraints can combine different types of checkers and solvers. A few examples will help to motivate the more general treatment of the next section.

The antecedents of conditional constraints in MLLP are required to contain only discrete variables, to accommodate a search algorithm that branches only on discrete variables. The algorithm could just as well branch on continuous variables by restricting them to successively smaller intervals, thus allowing the antecedents to contain continuous variables. This sort of branching is already done in some global optimization methods [111] as well as nonlinear solvers based on interval arithmetic [139]. Interval-based methods can be readily incorporated into the checker.

The conditional form of constraints would remain useful. It could still require constraints in the consequent that contain continuous variables to be linear inequalities, while allowing continuous variables in the antecedent to occur in any sort of context, such as a nonlinear function or a constraint that mixes discrete and continuous elements. The power of a linear solver would continue to be exploited.

Another possibility is to allow discrete variables into the consequent, not only within variable constants, subscripts, and index sets, but truly as variables to be manipulated by the solver. This is most easily done by letting $L_i(x, y)$ contain such specially-structured discrete constraints as Horn clauses as well as linear inequalities. If the discrete constraints had no continuous variables, a specialized logic processor such as unit resolution could be applied to the discrete portion to check for feasibility, while the LP solver minimizes cx subject to the linear constraints.

A third possibility is to allow nonlinear expressions into the consequent and replace the LP solver with a nonlinear programming solver. Most nonlinear solvers are incomplete when the feasible set is nonconvex, but this is acceptable if the solver becomes complete as one descends sufficiently far into the search tree. This can be accomplished by branching on continuous as well as discrete variables. As the interval constraints on the continuous variables become sufficiently tight, the feasible set described by $L_i(x, y)$ will become convex (for practical purposes) and the solver complete. Alternatively, one could require the feasible set for easy constraints to be convex.

Van Hentenryck and Michel [139, 99] recently developed a modeling language (Helios) for the nonlinear constraint-based solver Newton; the proposal here is a conditional modeling language that would integrate constraint solvers with optimizers.

3.3 A GENERAL MODELING LANGUAGE

Turning to the main issue at hand, how can a modeling language be designed in general to exploit the availability of a checker and a specialized solver? What are the essential features of such a language?

A checker can operate only by examining solutions that are generated by some other means. The solution algorithm must therefore proceed by enumerating solutions or partial solutions. The solver can be applied to a specially-structured constraint set whose composition, in the most general case, depends on the outcome of the checker.

3.3.1 The Basic Model

A solution procedure can be implemented if the model itself indicates how the solver's problem depends on the results of the checker. A natural way to do this is to suppose that the problem's constraints are conditional in form.

$$\begin{array}{ll} \min & f \\ \text{s.t.} & h_i \rightarrow E_i,\ i \in I, \\ & x \in D \end{array} \tag{3.12}$$

where D is the *domain*, the objective function f is a function of $x \in D$, h_i is a *hard* constraint, and E_i a collection of *easy* constraints. The domain D is the cross-product of the domains of the variables x, that is $D_{x_1} \times \ldots \times D_{x_n}$; an element x of D, $x \in D$ is a tuple $v_1 \times \ldots \times v_n$, where $v_i \subseteq D_{x_i}, 1 \leq i \leq n$. Each constraint is either satisfied or violated by a given $x \in D$.

In MLLP, the hard constraints can be any constraints containing discrete variables, the easy constraints are linear inequalities, and the domain D is $R^m \times D_1 \times \ldots \times D_n$. The general form (3.12) does not exclude the possibility that the same variable may occur in both the antecedent and consequent. It therefore encompasses variable constants, subscripts and index sets by allowing discrete variables to occur in the consequent.

For an element $x \in D$, let $E(x)$ be the union of all easy constraints E_i for which x satisfies h_i. Then x is a *feasible solution* if x satisfies $E(x)$, which may be empty and therefore trivially satisfied.

3.3.2 The Test Set Reformulation

It is usually impractical to enumerate complete solutions x. In MLLP, for example, this would be an enumeration of complete vectors (x, y) of continuous and discrete variables. It is therefore convenient to reformulate

the problem so that one can enumerate partial solutions, or more generally, *test sets* of solutions. The test sets are subsets of the domain D, not necessarily disjoint. Each test set T is defined by a collection $C(T)$ of *test set constraints*. In traditional MLLP, test sets are defined by partial assignments to the vector y of discrete variables. So $C(T)$ has the form $\{(y_{j_1}, \ldots, y_{j_k}) = (\alpha_1, \ldots, \alpha_k)\}$, where $\alpha_1 \ldots, \alpha_k$ are constants and $1 \leq k \leq n$. If the algorithm branches on continuous intervals, $C(T)$ would consist of bounds $u \leq x \leq v$ that become narrower as one descends into the search tree. The test set constraints must be *easy* constraints, because they will form part of the constraint set passed to the solver, albeit this is not an issue in MLLP, where the easy constraints contain no discrete variables.

Example 1 Suppose that the easy constraints are inequalities of the form $P(x) \leq 0$, where $P(x)$ is a convex polynomial in continuous variables $x = (x_1, \ldots, x_n)$. The hard constraints are discrete logic relations, plus inequalities of the form $Q(x) \leq 0$, where $Q(x)$ is any polynomial. A problem formulation in this context might be,

$$\begin{array}{ll} \min & (x_1 \Leftrightarrow 10)^2 + (x_2 \Leftrightarrow 10)^2 \\ \text{s.t.} & (x_1 x_2 \geq 1) \rightarrow (x_1^2 + x_2^2 \leq 98) \\ & (y_1 \neq y_2) \rightarrow x_1 + x_2 \leq 10 \\ & 0 \leq x_1, x_2 \leq 10, \quad y_1, y_2 \in \{1, 2\} \end{array}$$

If one branches by fixing discrete variables and imposing bounds on the continuous variables, then test sets have the form

$$T = [L_1, U_1] \times [L_2, U_2] \times D_1 \times D_2$$

where each $[L_j, U_j]$ is a closed interval and each D_j is $\{1\}$, $\{2\}$ or $\{1, 2\}$. The test set constraints have the form,

$$C(T) = \{L \leq x \leq U,\ (y_1, \ldots y_k) = (v_1, \ldots, v_k)\}$$

where $k \in \{0, 1, 2\}$ and each $v_j \in \{1, 2\}$. $\square$

A test set T *satisfies* a constraint if all the elements of T satisfy the constraint, and T *violates* the constraint if all elements of T violate it. A nonempty test set T is *determining* if it satisfies or violates any given hard constraint in the problem. Let $E(T)$ be the union of all easy constraint sets E_i for which T satisfies h_i. If T is determining, $E(T) = E(x)$ for all $x \in T$. In MLLP, a test set in which all of the discrete variables are fixed would be determining. There are generally other determining test sets as

well, because it is usually unnecessary to fix all of the discrete variables to determine whether the hard constraints are satisfied or violated. A test set collection is *exhaustive* if the union of its determining test sets is equal to the domain.

Example 2 Continuing with the problem of Example 1, the test set $[5,8] \times [5,8] \times \{1\} \times \{1\}$ satisfies constraint $x_1 x_2 \geq 1$ (because the product of the lower bounds of x_1 and x_2 equals 25), while it violates constraint $y_1 \neq y_2$. The test set is therefore determining.

The collection of all test sets is exhaustive because test sets of the form $[L_1, L_1] \times [L_2, L_2] \times \{v_1\} \times \{v_2\}$ are always determining. □

In optimization problems it is convenient to assume that each test set T is "closed" with respect to f. This means that one of the following is true: a) the minimum of $f(x)$ over $E(T) \cup C(T)$ exists, b) the minimum is unbounded, or c) $E(T) \cup C(T)$ is unsatisfiable. The *value* $f(T)$ of T corresponding to the three cases is a) the minimum value of $f(x)$, b) $\Leftrightarrow\infty$, or c) ∞.

A test set T is *feasible* if it is determining and $E(T) \cup C(T)$ is satisfiable. A test set is *optimal* if it is feasible and no feasible test set has a smaller value. If an exhaustive collection of closed test sets is given for the problem (3.12), the corresponding *test set formulation* is the problem of finding an optimal test set in the collection.

Example 3 For the problem in Example 1, the test set $[5,8] \times [5,8] \times \{1\} \times \{1\}$ is closed because $E(T) \cup C(T) = \{x_1^2 + x_2^2 \leq 98, 5 \leq x_1 \leq 8, 5 \leq x_2 \leq 8, (y_1, y_2) = (1,1)\}$ has a well-defined minimum value 18 at $(x_1, x_2, y_1, y_2) = (7,7,1,1)$ for objective function $(x_1 \Leftrightarrow 10)^2 + (x_2 \Leftrightarrow 10)^2$. It is also feasible and optimal. □

Theorem 1 *If (3.12) has an optimal solution x, then any given test set formulation of (3.12) has an optimal solution T containing x. Conversely, if a test set formulation of (3.12) has an optimal solution T, then (3.12) has an optimal solution $x \in T$.*

Proof. Suppose first that there is an optimal solution x for (3.12), and show that x belongs to an optimal test set T. Because the test sets are exhaustive, some determining T contains x. Because x satisfies $E(x)$ and T is determining, $E(x) = E(T)$ and x satisfies $E(T)$. So T is feasible. Furthermore, because x is optimal in (3.12), $f(x)$ is the value of T, which means T is optimal.

Now suppose that there is an optimal test set T, and show that some optimal solution x belongs to T. Because T is optimal and closed, some x in T minimizes f subject to $E(T) \cup C(T)$. Because T is determining, $E(T) = E(x)$, which means that x satisfies $E(x)$ and is therefore feasible in (3.12). Furthermore, x is optimal in (3.12). If it were not, some feasible $x' \in D$ would satisfy $f(x') < f(x)$, and by the above argument there would be a test set T' containing x' with $f(T') < f(T)$. □

3.3.3 Solving the Test Set Reformulation

One can solve a test set relaxation of (3.12) by enumerating an exhaustive sequence of test sets. In practice the enumeration usually takes the form of branching, but other schemes are possible.

If the branching algorithm is to work, the checker and solver must not only be correct, but they must become complete in an appropriate sense as one descends into the tree. The checker is *correct* if it finds T to satisfy (or violate) any given h_i *only if* T really does satisfy (or violate) h_i. The checker is *complete* for T if it finds T to satisfy (or violate) any given h_i *whenever* T really satisfies (or violates) h_i.

The solver is *correct* for a test set T if for any set E of easy constraints, the minimum value of f it finds, subject to $E \cup C(T)$, is an upper bound on the true minimum value, and the solution x (if any) it finds is feasible. The solver is *complete* for T if, for any E, the minimum value it finds is a global minimum over the feasible set defined by $E \cup C(T)$, and it finds an optimal solution x if one exists.

At what depth in the search tree should the checker and solver become complete? It is impractical to require that they become complete as soon as the test sets become determining. In MLLP, for example, a test set may become determining with only a few discrete variables are fixed, while further variables must be fixed before the checker can determine whether the hard constraints are satisfied or violated. One can suppose, however, that at a sufficient depth one reaches *elementary* test sets that are not only determining but for which the checker and solver are known to be complete. The elementary test sets are *exhaustive* if their union is equal to the domain. They should also be finite in number, to ensure that the search terminates. In MLLP, the elementary test sets might be defined by fixing all discrete variables. In nonlinear continuous programming, they may consist of intervals so small that any reasonable function can be assumed to be convex (even linear) on them.

Example 4 A finite collection of elementary test sets for the problem of Example 1 might be those in which the discrete variables are fixed and the

continuous variables confined to an interval of the form $[k\delta, (k+1)\delta]$, where k is an integer and $\delta > 0$ a small number. They define a fine grid over $\Re^2$. It is true that even an elementary test set such as $[1 \Leftrightarrow \delta, 1] \times [1, 1+\delta] \times \{1\} \times \{1\}$ neither satisfies nor violates $x_1 x_2 \geq 1$. This can be dealt with by making δ small enough so that calculations with $x_j = v$ give the same result as calculations with $x_j = v \pm \delta$. So the operation $x_1 x_2$ is, strictly speaking, defined as the output of a computational procedure with finite precision. The collection of elementary test sets is obviously exhaustive.

Given a test set $[L_1, U_1] \times [L_2, U_2] \times D_1 \times D_2$, the checker need only evaluate $L_1 L_2$ and $U_1 U_2$ to check satisfaction of $x_1 x_2 \leq 1$. The checker is clearly complete for elementary test sets. It is practical to use a nonlinear solver that is complete for all test sets, and in particular for elementary test sets, because the objective function and feasible set of the continuous optimization problem are convex. □

As in the case of MLLP, the algorithm proceeds by examining a sequence of test sets. Each iteration again consists of two steps.

- The checker tries to determine for each hard constraint h_i whether the current test set T satisfies it, violates it, or neither. It finds h_i to be *determined* if it determines that T satisfies it or violates it. The checker finds T to be determining if it finds every hard constraint to be determined. If T is elementary, the checker will necessarily find T to be determining, but T may be found determining even if it is not elementary.
- The solver minimizes f subject to $\bar{E}(T) \cup C(T)$, where $\bar{E}(T)$ is the union of all E_i for which the checker finds that T satisfies h_i. If T is elementary, the solver will be complete for T and will necessarily obtain the true optimal value and an optimal solution x if one exists. However, one may know that the solver is complete for T even when T is not elementary.

The algorithm now takes action based on the outcome of the two steps. The possibilities appear in Table 3.1. If T is found to be determining, there are two cases: the solver is known to be complete for T, in which case the search "backtracks," or it is not known to be complete, in which case the search "branches." In the former case, if the solver finds the current solution to be better than the incumbent, it records the current solution as the incumbent before backtracking.

If T is not found to be determining, there are two relevant outcomes. If the solver is known to be complete for T and fails to find a solution better than the incumbent, the search backtracks. Otherwise it branches.

		Result of Solver		
		Known to be complete for T		*Not known compl. for T*
		Opt. val. $< \bar{z}$	*Opt. val.* $\geq \bar{z}$	
Result of checker	*T found to be det.*	record inc.; backtrack	backtrack	branch
	T not found to be det.	branch	backtrack	branch

Table 3.1: Possible outcomes of each iteration of a search algorithm. $\bar{z}$ is the current upper bound on the optimal value.

"Backtracking" and "branching" are conceived here as generalizations of the ideas as they occur in tree search algorithms, but they can be any scheme for choosing the next test set that satisfies the conditions in the theorem below. The precise algorithm appears in Fig. 3.2.

```
Let Backtrack and Branch be procedures that choose the next
   test set from a finite and exhaustive collection of
   test sets, or the empty set when no further test sets
   can be chosen.
Let T be a nonempty initial test set.
Let z̄ be an upper bound on the optimal value, initially ∞.
While T is nonempty:
   If the solver is known to be complete for T:
      Let Ē(T) be the union of all constraint sets E_i for
         which the checker finds T to satisfy h_i.
      Let z be the minimum value of cx subject to Ē(T) ∪ C(T)
         found by the solver.
      If z < z̄ then
         If the checker finds T to be determining then
            Let z̄ = z.
            If z > −∞ then let T* = T.
            Call Backtrack.
         else call Branch to choose the next test set T.
      Else call Backtrack to choose the next test set T.
   Else call Branch to choose the next test set T.
If z̄ = ∞ then the problem is infeasible.
Else if z̄ > −∞ then T* is an optimal solution.
Else the problem is unbounded.
```

Figure 3.2: A generic solution algorithm for the test set reformulation.

Theorem 2 *Suppose that a search algorithm is defined on a finite collection of test sets and has the following properties.*

a) It never branches on the same test set twice.

b) If it backtracks on a test set, then it never checks a subset of it.

c) It terminates only when a test set is found to be feasible or no test set satisfies conditions (a) and (b).

d) The checker and solver are complete for an exhaustive collection of elementary test sets within the test set collection.

Then the algorithm is finite and solves the test set reformulation of (3.12) if and only if a solution exists.

Proof. The algorithm is clearly finite because it cannot check the same test set twice, and there are a finite number of test sets. It remains to show that it terminates with an optimal test set if and only if one exists.

Suppose first that some test set T^* is optimal with value z^*, and show that the algorithm discovers an optimal test set. Due to Theorem 1, T^* contains an optimal solution x^*. Because the elementary test sets are exhaustive, x^* belongs to some elementary test set T. Both T^* and T are determining, which means that $E(T^*) = E(x^*) = E(T)$. This and the fact that x^* satisfies $E(T^*) \cup C(T^*)$ imply that x^* satisfies $E(T) \cup C(T)$, which means that $E(T) \cup C(T)$ is satisfiable and T therefore feasible. Because T's value is the value of x^* and is therefore z^*, T is optimal. Now condition (c) implies that there are two cases.

(i) The algorithm checks T. Because T is elementary and the checker is complete, the checker finds T to be determining. Because T is feasible and the solver is complete, the solver obtains the optimal value z^*. The search therefore terminates with an incumbent solution that is T or a previously discovered test set with value z^*.

(ii) The algorithm backtracks on a superset T' of T. Let z be the optimal value obtained by the solver for T'. Then

$$\begin{aligned} z = &\min\{f \mid \bar{E}(T') \cup C(T')\} \leq \\ &\min\{f \mid E(T') \cup C(T')\} \leq \\ &\min\{f \mid E(T) \cup C(T)\} = z^*. \end{aligned}$$

The first equality is due to the fact that the solver was known to be complete for T', or else the search would not have backtracked. The first inequality follows from $\bar{E}(T') \subset E(T')$. The second inequality follows from $T \subset T'$. Thus $z \leq z^*$, and because z^* is the optimal value, $z = z^*$. The fact that the algorithm backtracked on T' implies that it either recorded T' as an incumbent solution or had already found a solution of equal value.

For the converse, suppose that the algorithm terminates with an incumbent solution T^*, and show that T^* is optimal. Because T^* is the incumbent

solution, the checker found T^* to be determining, and the solver found f to have a finite minimum value $\bar{z}$ subject to $\bar{E}(T^*) \cup C(T^*)$. So $\bar{E}(T^*) = E(T^*)$, which means that $\bar{z}$ is the minimum value of f subject to $E(T^*) \cup C(T^*)$ and is therefore the value of T^*. It remains to show that no test set has a value smaller than $\bar{z}$. If there were such a test set, then as just shown the algorithm would have discovered it and therefore could not have terminated with incumbent solution T^*. $\square$

The finiteness condition on the test set collection seems innocuous. Even if the test sets are defined by interval constraints on continuous variables, for example, one need only require that every test set constraint $u \leq x_j \leq v$ be of the form $p\delta \leq x_j \leq q\delta$, where $\delta > 0$ is a small number and p, q are integers. If global lower and upper bounds can be placed on the variables, the test set is finite.

As in the case of MLLP, the generic algorithm of Fig. 3.2 can incorporate a stronger relaxation. Rather than minimize f subject to $\bar{E}(T) \cup C(T)$, the solver can find the minimum value of f subject to $R(T) \cup C(T)$. $R(T)$ is any set of easy constraints such that this minimum value is a lower bound on the optimal value of (3.12), with $C(T)$ added to the constraint set of the latter.

3.4 CONCLUSION

We have shown how a model with conditional constraints can dictate how a checker and a solver may be combined in a generic branching algorithm. To accomplish this, we analyzed the precise structure of a branching procedure that involves both constraint checking and optimization. The process is unexpectedly complex, as it involves checking for satisfaction when the variables are assigned a subset of the domain (test set) rather than a point value. This test set may neither satisfy nor violate a given constraint, and even if it does one or the other, the checker may be unable to deduce that it does. It is therefore only under fairly restrictive conditions that a simple branching procedure is known to terminate with a solution.

These conditions take the form of properties that possible test sets must have. The collection of test sets must contain a finite subcollection of elementary test sets that cover the entire solution space. The elementary test sets must be simple enough so that the checker can always determine whether a given one of them satisfies or violates constraints in the antecedents of the conditionals (the hard constraints). The solver must be able to find a globally optimal solution subject to constraints consisting of a) constraints in the enforced consequents of the conditions, and b) constraints that define an elementary test set. Finally, the search procedure must have certain basic characteristics of a branching search.

Once these conditions are met, a model in conditional form can be seen as indicating how a checker and a solver may work together to solve the model. In particular, if the checker applies constraint satisfaction procedures and the solver applies linear or nonlinear programming, then these methods are combined in a principled way.

Note, however, that it is not simply a model in conditional form that has this ability. It is a model in conjunction with a defined class of easy constraints, a class of hard constraints, and a collection of test sets defined by the branching mechanism. All of these elements must satisfy the conditions just enumerated.

A modeling language following the proposed schema could allow the user to choose the elements that apply to the model at hand. It could provide a menu of:

1. Several types of hard constraints, each with its own syntax rules. The user would specify at the outset which hard constraint syntax would be used. The solution algorithm would use a checker designed for the chosen syntax.

2. Several types of easy constraints. Again the user would specify which one is to be used, and an associated solver would be invoked in the solution phase.

3. Several types of test set constraints. The user must choose a type that satisfies the chosen syntax rules for easy constraints. The branching or search procedure would depend on the type of test sets used.

4. A general conditional form for constraints of the model, in which the antecedents must be hard constraints and the consequents sets of easy constraints.

5. A definition of what is to be an elementary test set, chosen so that the elementary test sets are finite in number, and so that the checker and solver are complete for them.

6. Optionally, a means of recognizing when the solver is complete for test sets that are not necessarily elementary.

The discussion here has been restricted to situations in which there is a single checker and a single solver. A natural question is whether a model might accommodate several types of hard and easy constraints, and whether a single search procedure can integrate the corresponding checkers and solvers. This issue is left to future research.

On Integrating Constraint Propagation and Linear Programming for Combinatorial Optimization

John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson and Hak-Jin Kim

Linear programming and constraint propagation are complementary techniques with the potential for integration to benefit the solution of combinatorial optimization problems. Attempts to combine them have mainly focused on incorporating either technique into the framework of the other — traditional models have been left intact. We argue that a rethinking of our modeling traditions is necessary to achieve the greatest benefit of such an integration. We propose a declarative modeling framework in which the structure of the constraints indicates how LP and CP can interact to solve the problem.

4.1 INTRODUCTION

Linear programming (LP) and constraint propagation (CP) are techniques from different fields that tend to be used separately in integer programming (IP) and constraint (logic) programming (CLP), respectively. They have the

potential for integration to benefit the solution of combinatorial optimization problems. Yet only recently have attempts been made at combining them.

IP has been successfully applied to a wide range of problems, such as capital budgeting, bin packing, crew scheduling and traveling salesman problems. CLP has in the last decade been shown to be a flexible, efficient and commercially successful technique for scheduling, planning and allocation. These problems usually involve permutations, discretization or symmetries that may result in large and intractable IP models.

Both CLP and IP rely on branching to enumerate regions of the search space. But within this framework they use dual approaches to problem solving: inference and search. CLP emphasizes inference in the form of constraint propagation, which removes infeasible values from the variable domains. It is not a search method, i.e., an algorithm that examines a series of complete labellings until it finds a solution. IP, by contrast, does exactly this. It obtains complete labellings by solving linear programming relaxations of the problem in the branching tree.

IP has the advantage that it can generate cutting planes (inequalities implied by the constraint set) that strengthen the linear relaxation. This can be a powerful technique when the problem is amenable to polyhedral analysis. But IP has the disadvantage that its constraints must be expressed as inequalities (or equations) involving integer-valued variables. Otherwise the linear programming relaxation is not available. This places a severe restriction on IP's modeling language.

In this paper we argue that the key to effective integration of CP and LP lies in the design of the modeling language. We propose a language in which conditional constraints indicate how CP and LP can work together to solve the problem.

We begin, however, by reviewing efforts that have hitherto been made toward integration.

4.2 PREVIOUS WORK

Several articles compare CLP and IP [132, 41]. They report experimental results that illustrate some key properties of the techniques: IP is very efficient for problems with good relaxations, but it suffers when the relaxation is weak or when its restricted modeling framework results in large models. CLP, with its more expressive constraints, has smaller models that are closer to the problem description and behaves well for highly constrained problems, but it lacks the "global perspective" of relaxations.

Some attempts have been made at integration. Early out was [18] in which the idea is explored of coupling CP and LP solvers with bounds propagation and fixed variables. In [127], CP is used along with LP relaxations in a single search tree to prune domains and establish bounds. A node can fail either because propagation produces an empty domain, or the LP relaxation is infeasible or has an optimal value that is worse than the value of the optimal solution (discussed below). A systematic procedure is used to create a "shadow" MIP model for the original CLP model. It includes reified arithmetic constraints (which produce big-M constraints, illustrated below) and `alldifferent` constraints. The modeler may annotate constraints to indicate which solver should handle them — CP, LP or both.

Some research has been aimed at incorporating better support for symbolic constraints in IP. [71, 72] show how disequalities ($X_i \neq X_j$) can be handled (more) efficiently in IP solvers. Further, they give a linear modeling of the `alldifferent` constraint.

Bockmayr and Kasper propose an interesting framework in [20] for combining CLP and IP, in which several approaches to integration or synergy are possible. They investigate how symbolic constraints can be incorporated into IP much as cutting planes are. They also show how a linear system of inequalities can be used in CLP by incorporating it as a symbolic constraint. They also discuss a closer integration in which both linear inequalities and domains appear in the same constraint store.

4.3 CHARACTERIZATION

We start with a basic characterization of CLP and IP.

4.3.1 Constraint (Logic) Programming

In Finite Domain CLP each integer variable x_i has an associated *domain* D_i, which is the set of possible values this variable can take on in the (optimal) solution. The cartesian product of the domains, $D_1 \times \ldots \times D_n$, forms the solution space of the problem. This space is finite and can be searched exhaustively for a feasible or optimal solution, but to limit this search CP is used to infer infeasible solutions and prune the corresponding domains. From this viewpoint, CP operates on the set of possible solutions and narrows it down.

4.3.2 Integer Programming

In contrast to CLP, IP does not maintain and reduce a set of solutions defined by variable domains. It generates a series of complete labellings, each obtained at a node of the branching tree by solving a relaxation of the

problem at that node. The relaxation is usually constructed by dropping some of the constraints, notably the integrality constraints on the variables, and perhaps by adding valid constraints (cutting planes) that make the relaxation tighter. In a typical application the relaxation is rapidly solved to optimality with a linear programming algorithm.

If the aim is to find a feasible solution, branching continues until the solution of the relaxation happens to be feasible in the original problem (in particular, until it is integral). Relaxations therefore provide a heuristic method for identifying solutions. In an optimization problem, relaxation also provides bounds for a branch-and-bound search. At each node of the branching tree, one can check whether the optimal value of the relaxation is better than the value of the best feasible found so far. If not, there is no need to branch further at that node.

The dual of the LP relaxation can also provide useful information, perhaps by fixing some integer variables or generating additional constraints (nogoods) in the form of "Benders cuts." [17, 63]. We will see how the latter can be exploited in an integrated framework.

4.3.3 Comparison of CP and LP

CP can accelerate the search for a solution by

- reducing variable domains (and in particular by proving infeasibility),
- tightening the linear relaxation by adding bounds and cuts in addition to classical cutting planes, and
- eliminating search of symmetric solutions, which are often more easily excluded by using symbolic constraints.

LP can enhance the solver by

- finding feasible solutions early in the search by "global reasoning," i.e., solution of an LP relaxation,
- similarly providing stronger bounds that accelerate the proof of optimality, and
- providing reasons for failure or a poor solution, so as to produce nogoods.

4.4 MODELING FOR HYBRID SOLVERS

The approaches taken so far to integration of CP and LP are (a) to use both models in parallel, and (b) to try to incorporate one within the other. The more fundamental question of whether a *new* modeling framework should be used has not yet been explored in any depth. The success of (a) depends on the strength of the links between the models and to what degree the overhead of having two models can be avoided. Option (b) is limited in what it can achieve. The high-level symbolic constraints of CLP cannot directly be applied in an IP model, and the same holds for attempts to use IP's cutting planes and relaxations in CLP.

We will use a simple multiple-machine scheduling problem to illustrate the advantages of a new modeling framework. Assume that we wish to schedule n tasks for processing on as many as n machines. Each machine m runs at speed r_m. The objective is to minimize the total fixed cost of the machines we use. Let R_j be the release time, P_j the processing time for speed $r_m = 1$ and D_j the deadline for task j. Let C_m be the fixed cost of using machine m and t_j the start time of task j.

We first state an IP model, which uses 0–1 variables x_{ij} to indicate the sequence in which the jobs are processed. Let $x_{ij} = 1$ if task i precedes task j, $i \neq j$, on the same machine, with $x_{ij} = 0$ otherwise. Also let 0–1 variable $y_{mj} = 1$ if task j is assigned to machine m, and 0–1 variable $z_m = 1$ if machine m is used. Then an IP formulation of this problem is,

$$
\begin{aligned}
\min \quad & \sum_m C_m z_m & & \\
\text{s.t.} \quad & z_m \geq y_{mj}, & \forall m, j, & \qquad (4.1)\\
& \sum_m y_{mj} = 1, & \forall j, & \qquad (4.2)\\
& t_j + \sum_m \frac{P_j}{r_m} y_{mj} \leq D_j, & \forall j, & \\
& R_j \leq t_j, & \forall j, & \\
& t_i + \sum_m \frac{P_i}{r_m} y_{mi} \leq t_j + M(1 \Leftrightarrow x_{ij}), & \forall i \neq j, & \qquad (4.3)\\
& x_{ij} + x_{ji} \geq y_{mi} + y_{mj} \Leftrightarrow 1, & \forall m, i < j, & \qquad (4.4)\\
& z_m,\, y_{mj},\, x_{ij} \in \{0,1\},\ t_j \geq 0, & \forall m, i, j. &
\end{aligned}
$$

Constraint (4.3) is a "big-M" constraint. If M is a sufficiently large number, the constraint forces task i to precede task j if $x_{ij} = 1$ and has no effect otherwise.

A CLP model for the same problem is

$$
\begin{array}{lll}
\min & \displaystyle\sum_m C_m z_m & \\
\text{s.t.} & \text{if } m_i = m_j \text{ then} & \\
& \quad (t_i + \dfrac{P_i}{r_{m_i}} \leq t_j) \vee (t_j + \dfrac{P_j}{r_{m_j}} \leq t_i), & \forall i, j, \qquad (4.5) \\
& t_j + \dfrac{P_j}{r_{m_j}} \leq D_j, & \forall j, \\
& R_j \leq t_j, & \forall j, \\
& \text{if } \texttt{atleast}(m,[m_1, \ldots, m_n], 1) & \\
& \quad \text{then } z_m = 1 \text{ else } z_m = 0, & \forall m, \qquad (4.6) \\
& m,\, m_j \in \{1, \ldots, n\},\ t_j \geq 0, & \forall j,
\end{array}
$$

where m_j is the machine assigned to task j. The CLP model has the advantage of dispensing with the doubly-subscripted 0–1 variables x_{ij} and y_{mi}, which are necessary in IP to represent permutations and assignments. This advantage can be pronounced in larger problems. A notorious example is the progressive party problem (Smith et al., 1995), whose IP model requires an enormous number of multiply-subscripted variables.

The IP model has the advantage of a useful linear programming relaxation, consisting of the objective function, constraints (4.1)–(4.2), and bounds $0 \leq y_{mj} \leq 1$. The 0–1 variables y_{mj} enlarge the model but compensate by making this relaxation possible. However, the IP constraints involving the permutation variables x_{ij} yield a very weak relaxation and needlessly enlarge the LP relaxation.

Somehow we must combine the succinctness of the CLP model with an ability to create a relaxation from that portion of the IP model that has a useful relaxation. To do this we propose taking a step back to investigate how one can design a model to suit the solvers rather than adjust the solvers to suit the traditional models.

4.5 MIXED LOGICAL/LINEAR PROGRAMMING

We begin with the framework of Mixed Logical / Linear Programming (MLLP) proposed in [75, 81, 82]. It writes constraints in the form of conditionals that link the discrete and continuous elements of the problem. A model has the form

$$
\begin{array}{lll}
\min & cx & \qquad (4.7) \\
\text{s.t.} & h_i(y) \rightarrow A^i x \geq b^i, \ \ i \in I, & \\
& x \in R^n, y \in D, &
\end{array}
$$

where y is a vector of discrete variables and x a vector of continuous variables. The antecedents $h_i(y)$ of the conditionals are constraints that can be treated with CP techniques. The consequents are linear inequality systems that can be inserted into an LP relaxation.

A linear constraint set $Ax \geq b$ which is enforced unconditionally may be so written for convenience, with the understanding that it can always be put in the conditional form $(0 = 0) \rightarrow Ax \geq b$. Similarly, an unconditional discrete constraint h can be formally represented with the conditional $\neg h \rightarrow (1 = 0)$.

The absence of discrete variables from the objective function will be useful algorithmically. Costs that depend on discrete variables can be represented with conditional constraints. For example, the objective function $\sum_j c_j x_j$, where $x_j \in \{0, 1\}$, can be written $\sum_j z_j$ with constraints $(x_j = 1) \rightarrow (z_j = c_j)$ and $z_j \geq 0$ for all j.

A useful modeling device is a *variable subscript*, i.e., a subscript that contains one or more discrete variables. For example, if c_{jk} is the cost of assigning worker k to job j, the total cost of an assignment can be written $\sum_j c_{jy_j}$, where y_j is a discrete variable that indicates the worker assigned job j. The value of c_{jy_j} is in effect a function of $y = (y_1, \ldots .y_n)$ and can be written $g_j(y)$, where function g_j happens to depend only on y_j. The MLLP model can incorporate this device as follows:

$$\begin{array}{ll} \min & cx \\ \text{s.t.} & h_i(y) \rightarrow L_i(x, y), \quad i \in I, \\ & x \in R^n,\ y \in D, \end{array}$$

where

$$L_i(x, y) = \sum_{k \in K_i(y)} a_{ik}(y) x_{j_{ik}(y)} \geq b_i(y).$$

Note that the model also allows for a summation taken over a *variable index set* $K_i(y)$, which is a set-valued function of y, as well as real-valued "variable constants" $b_i(y)$.

Models of this sort can in principle be written in the more primitive form (4.7) by adding sufficiently many conditional constraints. For example, the constraint $z \geq \sum_j c_{jy_j}$ can be written $z \geq \sum_j z_j$, if the following constraints are added to the model

$$(y_j = k) \rightarrow (z_j = c_{jk}), \quad \text{all } j, k \in \{1, \ldots n\},$$

where each $y_j \in \{1, \ldots, n\}$. It is preferable, however, for the solver to process variables subscripts and index sets directly.

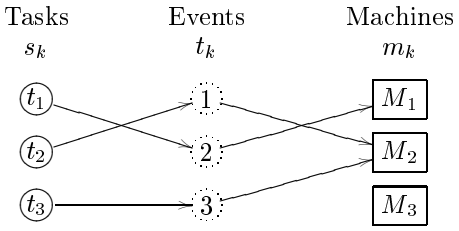
4.5.1 The Solution Algorithm

An MLLP problem is solved by branching on the discrete variables. The conditionals assign roles to CP and LP: CP is applied to the discrete constraints to reduce the search and help determine when partial assignments satisfy the antecedents. At each node of the branching tree, an LP solver minimizes cx subject to the inequalities $A^i x \geq b^i$ for which $h_i(y)$ is determined to be true. This delayed posting of inequalities leads to small and lean LP problems that can be solved efficiently. A feasible solution is obtained when the truth value of every antecedent is determined, and the LP solver finds an optimal solution subject to the enforced inequalities.

Computational tests reported in [81] suggest that an MLLP framework not only has modeling advantages but can often permit more rapid solution of the problem than traditional MILP solvers. However, a number of issues are not addressed in this work, including: (a) systematic implementation of variable subscripts and index sets, (b) taking full advantage of the LP solution at each node, and (c) branching on continuous variables and propagation of continuous constraints.

4.5.2 An Example

We can now formulate the multiple machine scheduling problem discussed earlier in an MLLP framework. Let k index a sequence of *events*, each of which is the start of some task. In the following model we will focus on the events, using mappings from events to tasks and events to machines, respectively.



Variable t_k will now be the start time of event k, s_k the task that starts, and m_k the machine to which it is assigned. The formal MLLP model is,

$$\begin{aligned}
\min \quad & \sum_m f_m \\
\text{s.t.} \quad & (m_k = m_l) \rightarrow (t_k + \frac{P_{s_k}}{r_{m_k}} \leq t_l), && \forall k < l, \\
& t_k + \frac{P_{s_k}}{r_{m_k}} \leq D_{s_k}, && \forall k, \\
& R_{s_k} \leq t_k, && \forall k, \\
& \texttt{alldifferent}\{s_1, \ldots, s_n\}, \\
& f_{m_k} = C_{m_k},\ f_m \geq 0, && \forall k, m.
\end{aligned}$$

The model shares CLP's succinctness by dispensing with doubly-subscripted variables. To obtain the relaxation afforded by IP, we can simply add the objective function and constraints (4.1)–(4.2) to the model, and link the variables y_{mi} to the other variables logically in (4.10).

$$\begin{aligned}
\min \quad & \sum_m C_m z_m \\
\text{s.t.} \quad & (m_k = m_l) \rightarrow (t_k + \frac{P_{s_k}}{r_{m_k}} \leq t_l), && \forall k < l, \\
& t_k + \frac{P_{s_k}}{r_{m_k}} \leq D_{s_k}, && \forall k, \\
& R_{s_k} \leq t_k, && \forall k, \\
& \texttt{alldifferent}\{s_1, \ldots, s_n\}, \\
& z_m \geq y_{mj}, && \forall m, j, && (4.8) \\
& \sum_m y_{mj} = 1, && \forall j, && (4.9) \\
& y_{m_k s_k} = 1, && \forall k. && (4.10)
\end{aligned}$$

The relaxation now minimizes $\sum_m C_m z_m$ subject to (4.8), (4.9), $0 \leq y_{mj} \leq 1$, and $y_{mj} = 1$ for all y_{mj} fixed to 1 by (4.10). One can also add a number of additional valid constraints involving the y_{mj}'s and the t_k's.

4.5.3 A Perspective on MLLP

The framework for integration described in [20] provides an interesting perspective on MLLP. The CLP literature distinguishes between *primitive* and *nonprimitive* constraints. Primitive constraints are "easy" constraints for which there are efficient (polynomial) satisfaction and optimization procedures. They are maintained in a *constraint store*, which in finite-domain

CLP consists simply of variable domains. Propagation algorithms for nonprimitive constraints retrieve current domains from the store and add the resulting smaller domains to the store.

In IP, linear inequalities over continuous variables are primitive because they can be solved by linear programming. The integrality conditions are (the only) nonprimitive constraints.

Bockmayr and Kasper propose two ways of integrating LP and CP. The first is to incorporate the LP part of the problem into a CLP framework as a nonprimitive constraint. Thus LP becomes a constraint propagation technique. It accesses domains in the form of bounds from the constraint store and add new bounds obtained by minimizing and maximizing single variable.

A second approach is to make linear inequalities primitive constraints. The constraint store contains continuous inequality relaxations of the constraints but excludes integrality conditions. For example, discrete constraints $x_1 \vee x_2$ and $\neg x_1 \vee x_2$ could be represented in the constraint store as inequalities $x_1 + x_2 \geq 1$ and $(1 \Leftrightarrow x_1) + x_2 \geq 1$ and bounds $0 \leq x_j \leq 1$. If constraint propagation deduced that x_2 is true, the inequality $x_2 \geq 1$ would be added to the store. This is an instance of what has long been known as "preprocessing" in MILP, which can therefore be viewed as a special case of this second kind of integration.

MLLP is a third type of integration in which two constraint stores are maintained (see Figure 4.1). A classical finite domain constraint store,

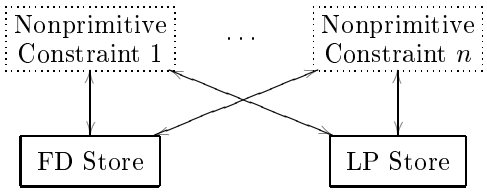


Figure 4.1: Constraint stores and nonprimitive constraints in MLLP

S_{FD}, contains domains, and the LP constraint store, S_{LP}, contains linear inequalities and bounds. The nonprimitive constraints can access and add to both constraint stores. Since only domain constraints $x_i \in D_i$ exist in the FD store, integrality constraints can remain therein as primitive constraints. There are no continuous variables in the CP store and no discrete variables in the LP store. The conditional constraints of MLLP act as the prime inference agents connecting the two stores, reading domains of the CP store and adding inequalities to the LP store (Figure 4.2).

In the original MLLP scheme, the conditionals are unidirectional, in the sense that they infer from S_{FD} and post to S_{LP} and not vice-versa. This is because the solution algorithm branches on discrete variables. As the discrete domains are reduced by branching, the truth value of more antecedents is inferred by constraint propagation, and more inequality constraints are posted. However, conditionals in the opposite direction could also be used if one branched on continuous variables by splitting intervals. The antecedents would contain continuous numerical constraints (not necessarily linear inequalities), and the consequents would contain primitive discrete constraints, i.e., restrictions on discrete variable domains. The truth value of the antecedents might be inferred using interval propagation.

We will next give two more examples of nonprimitive constraints in MLLP; a generalized version of the `element` constraint for handling variable subscripts, and a constraint which derives nogoods from S_{LP}.

4.5.4 Variable Subscripts

As seen before, MLLP provides variable subscripts as a modeling component, but an expansion to conditional constraints is in most cases not tractable. Instead a nonprimitive constraint, or inference agent, can be designed to handle variable subscripts more efficiently.

There are basically two cases in which a variable subscript can occur — in a discrete constraint or in a continuous inequality. In the former case it can either be a vector of constants or a vector of discrete variables; both of these correspond to the traditional use of the `element/3` [96] constraint found in all major CP systems and libraries (e.g. [49, 51]). This constraint takes the form $\texttt{element}_{FD}\texttt{(}I\texttt{, [}X_1, \ldots, X_n\texttt{], }Y\texttt{)}$, where I is an integer variable with domain $D_I = \{1, \ldots, n\}$, indexing the list, and $Y = X_I$.

Here we will consider the second case, $\texttt{element}_{LP}\texttt{(}I\texttt{, [}X_1, \ldots, X_n\texttt{], }Y\texttt{)}$, where I is still a discrete, indexing variable, but x_i and Y are *continuous* variables or constants. Propagating this constraint can be done almost as before. Let the interval $[\min(x_i), \max(x_i)]$ be the domain of x_i. Then upon change of the domain of I, we can compute

$$\begin{aligned}\min &= \{\min(x_i) | i \in D_I\} \\ \max &= \{\max(x_i) | i \in D_I\}\end{aligned}$$

reading D_I from the S_{CP} and the adding new bounds

$$\min \leq Y \leq \max$$

to S_{LP}. Similarly, bounds of Y can be used to prune D_I. (Stronger bounds for variables in LP can be obtained by minimizing and maximizing the

variable subject to the linear inequalities in S_{LP}, which for some cases might be beneficial.)

The important point here is not the details of how we can propagate this constraint, but rather to exemplify how an inference agent can naturally access both constraint stores.

4.5.5 Infeasible LP

When the LP is infeasible, any dual solution specifies an infeasible linear combination of the constraint set. For each conditional constraint $y_i \rightarrow A^i x \geq b^i, i \in I$ where some $ax \geq b \in A^i x \geq b^i$ has a corresponding nonzero dual multiplier, we can form the logical constraint

$$\bigvee_{i \in I} \neg y_i$$

This nogood [135] must be satisfied by any solution of the problem, because its corresponding set of linear inequalities forms an infeasible combination. This scheme can naturally be encapsulated within a nonprimitive constraint, `nogood`, reading from S_{LP} and writing to S_{FD}. This agent can collect, merge and maintain no-goods for any combination of nonzero dual values in any infeasible LP node, and can infer primitive and nonprimitive constraints which will strengthen S_{FD}. A related use of the infeasible combination has previously been explored in the context of intelligent backtracking [45]. Figure 4.2 shows our `nogood` constraint and the other basic nonprimitive constraints of MLLP.

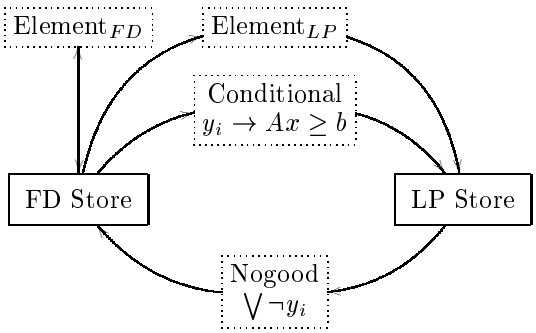


Figure 4.2: Nonprimitive constraints in MLLP

4.5.6 Feasible LP

In IP, the solution of the relaxation provides a complete labelling of the variables. This sort of labelling is not immediately available in MLLP,

because the relaxation involves only continuous variables. However, the solution $\bar{x}$ of a feasible relaxation can be heuristically extended to a complete labelling $(\bar{x}, \bar{y})$ that may satisfy the constraints. (Because y does not occur in the objective function, its value will not affect the optimal value of the problem.) Given any conditional $h_i(y) \rightarrow A^i x \geq b^i$, $h_i(\bar{y})$ must be false if $A^i \bar{x} \not\geq b^i$, but it can be true or false if $A^i \bar{x} \geq b^i$. One can therefore employ a heuristic (or even an exhaustive search) that tries to assign values $\bar{y}_j$ to the y_j's from their current domains so as to falsify the antecedents that must be false.

4.6 CONCLUSION

LP and CP have long been used separately, but they have the potential to be integrated as complementary techniques in future optimization frameworks. To do this fully and in general, the modeling traditions of mathematical programming and constraint programming also must be integrated. We propose a unifying modeling and solution framework that aims to do so. Continuous and discrete constraints are naturally combined using conditional constraints, allowing a clean separation and a natural link between constraints amenable to CP and continuous inequalities efficiently handled by LP.

Mixed Global Constraints and Inference in Hybrid CLP–IP Solvers

Greger Ottosson, Erlendur S. Thorsteinsson and John N. Hooker

The complementing strengths of Constraint (Logic) Programming (CLP) and Mixed Integer Programming (IP) have recently received significant attention. Although various optimization and constraint programming packages at a first glance seem to support mixed models, the modeling and solution techniques encapsulated are still rudimentary. Apart from exchanging bounds for variables and objective, little is known of what constitutes a good hybrid model and how a hybrid solver can utilize the complementary strengths of inference and relaxations. This paper adds to the field by identifying constraints as the essential link between CLP and IP and introduces an algorithm for bidirectional inference through these constraints. Together with new search strategies for hybrid solvers and cut-generating mixed global constraints, solution speed is improved over both traditional IP codes and newer mixed solvers.

5.1 INTRODUCTION

In this paper we continue exploring the integration of constraint programming and mathematical programming, specifically Constraint Propagation (CP) and Linear Programming (LP), extending our previous work [75, 81, 82, 84, 85]. In particular, we examine in more detail how to model for a hybrid solver and how to solve hybrid models, inter alia by giving specific examples. We also provide benchmarks for a production planning problem. The main contributions of this paper are:

- Mixed global constraints connecting CP and LP:
  - Variable subscripts and the compilation of variable subscripts in continuous functions, i.e., an extension of the `element` constraint to the continuous domain. We also describe its relation to disjunctive programming.
  - A mixed global constraint for semi-continuous piecewise linear functions.
- A scheme for bidirectional inference between CP and LP, i.e., between the Finite Domain constraint store (FD store) and the Linear Programming constraint store (LP store) [84].
- New search strategies for hybrid models.

The last few years have seen increasing interest and effort in the integration of constraint programming and mathematical programming. The main objective of such an integration is to take advantage of both the inference through CP and the (continuous) relaxations through LP, in order to reduce the size of the search tree.

The key decisions to be made for integrating Constraint (Logic) Programming (CLP) and Integer Programming (IP) are the (a) models, (b) inference, (c) relaxations, and, (d) search and branching strategies to use. For example, [20] introduces a framework with a combined constraint store and symbolic constraints that produce cutting planes; [74] combines two different models in two synchronized search trees; and, [127] automatically produces and updates a shadow copy of a CLP model on the continuous side, with constraint propagation and linear relaxations in a single search tree. A common feature of these methods is to communicate bounds, as introduced in [18].

In [84], we advocate to use neither the CLP nor the IP model but rather to model specifically for the hybrid solver, roughly separating the problem into a discrete part (FD store) and a continuous part (LP store). This achieves domain reduction on the FD store (constraint propagation), inference from the FD store to the LP store (bounds and cutting planes) and a natural relaxation (the LP relaxation).

Missing in previous research was inference from the LP store to the FD store, causing the communication to be mainly unidirectional. Constraint propagation can not be done effectively from the LP store to the FD store, since an LP solver only gives a single solution to the current problem, so we are not drawing inference from a set which contains all the possible solutions as in CLP. The bounds provided by LP are usually weak and too costly to improve. We remedy this by adding inference from the LP solution to the FD store.

This paper is organized as follows. This section laid out the history of efforts in integrating CLP and IP. Section 5.2 introduces our framework, Mixed Logical/Linear Programming (MLLP). In Sec. 5.3 we expand this framework with mixed global constraints, taking variable subscripts and piecewise linear functions as specific examples. Section 5.4 focuses on algorithms and rules for inference and branching strategies. In Sec. 5.5 we introduce a production planning problem and then compare MLLP computationally with other approaches in Sec. 5.5.3. Finally, Sec. 5.6 summarizes our results.

5.2 MIXED LOGICAL/LINEAR PROGRAMMING (MLLP)

To lay the basis for the subsequent discussion, we recapitulate the basic framework of MLLP, proposed in [75, 81, 82, 84, 85]. In that framework, constraints are in the form of conditionals that link the discrete and continuous elements of the problem. An MLLP model has the form

$$\begin{aligned} \min\ & cx \\ \text{s.t.}\ & h_i(y) \rightarrow A^i x \geq b^i,\ \ i \in I, \\ & x \in \mathbb{R}^n,\ y \in D, \end{aligned} \tag{5.1}$$

where y is a vector of discrete variables and x a vector of continuous variables. The antecedents $h_i(y)$ of the conditionals are constraints that can be treated with CP techniques. The consequents are linear inequality systems that form an LP relaxation.

A linear constraint set $Ax \geq b$ which is enforced unconditionally may be so written for convenience, with the understanding that it can always be put in the conditional form $\mathrm{T} \rightarrow Ax \geq b$. Similarly, an unconditional discrete constraint h can be formally represented with the conditional $\neg h \rightarrow (0x = 1)$.

An MLLP problem is solved by branching on the discrete variables. The conditionals assign roles to CP and LP: CP is applied to the discrete constraints to reduce the search and help determine when partial assignments satisfy the antecedents. At each node of the branching tree an LP solver minimizes cx subject to the inequalities $A^i x \geq b^i$ for which $h_i(y)$ is determined to be true (entailed). This delayed posting of inequalities leads to small and lean LP problems that can be solved efficiently. A feasible solution is obtained when the truth value of every antecedent is determined (entailed or disentailed) and the LP solver finds an optimal solution subject to the enforced inequalities.

5.3 MIXED GLOBAL CONSTRAINTS

The need for global constraints, such as `alldifferent`, has been recognized for quite some time in constraint programming. One reason they were introduced is that they allow the modeler to represent a problem in a more "natural" and compact manner, i.e., they extend the expressiveness of the modeling language. Also, they open up the possibility to include structure specific propagation into a general solver and are thus extremely important for the efficiency of the solver.

Mathematical programming has still not seen the advantage of global constraints to the same extent. Modeling languages for LP/IP, such as AMPL [59], do include some constructs to aid the modeler in writing compact and easy-to-understand models. Those constructs, however, have to be transformed into linear inequalities before being sent to an LP/IP solver and the structural information is lost in the process. Although the structure may be known to the modeler it can not be communicated to the solver and the solver has to find and recognize the structure on its own to be able to apply logical processing to it, resulting in less efficient solution algorithms than what would have been possible. Recently though some work has been done on this topic [20].

In a framework such as MLLP, *mixed* global constraints serve both as a modeling tool and a way to exploit structure in the solution process. Mixed global constraints can be written in the form (5.1) as conditionals, analogous to global constraints in CLP, but improve the solution process by improving the propagation. This will be illustrated for variable subscripts and piecewise linear functions.

5.3.1 Variable Subscripts in Linear Constraints

Variable subscripts, i.e., subscripts that contain one or more discrete variables, are a very useful modeling device. For example, if c_{jk} is the cost of assigning worker k to job j, the total cost of an assignment can be written $\sum_j c_{jy_j}$, where y_j is a discrete variable indicating the worker assigned to job j. The value of c_{jy_j} can, however, not be determined when the model is compiled, as it hinges on the value of the variable y_j, and thus has to be deferred to the solver in some form.

A variable subscripted expression can, in principle, be written in the more primitive form (5.1) of conditional constraints. For example, the constraint $z \geq \sum_j c_{jy_j}$ can be written $z \geq \sum_j z_j$, if the following conditional constraints are added to the model,

$$(y_j = k) \rightarrow (z_j = c_{jk}), \quad \forall (j,k) \in \{1, \ldots, n\} \times \{1, \ldots, m\}.$$

It is preferable, however, that the solver process variable subscripts directly, to improve the inference, as will become more clear in what follows.

A constraint called (discrete) `element`, which mimics array lookup, has been used for a long time in the CLP community to represent variable subscript [96]. The expression $z = x_y$, where z and y are discrete variables and $x_1, \ldots, x_n$ are discrete variables (or constants), is equivalent to

$$\texttt{element}(y, [x_1, \ldots, x_n], z),$$

given that the domain of y is $D_y = \{1, \ldots, n\}$. Propagation for this constraint is usually [hyper]arc- or bounds-consistency if x is an array of constants [variables].

We expand on this construct by allowing variable subscripts to appear in continuous linear functions (constraints or objective function). The variable subscripts are presented to the solver in the following way: Any term with a variable subscript is replaced by an artificial variable in the LP solver, which is then constrained by introducing one or more *mixed* `element` constraints. For example, the linear constraints

$$d_y x_y + \sum_{i \in S} c_{i,y} = 42, \quad d_y x_y \geq 0,$$

where $S = \{1, \ldots, m\}$ and $y \in \{1, \ldots, n\}$, are compiled into

$$z_1 + \textstyle\sum_{i \in S} w_i = 42, \quad z_2 \geq 0,$$

$$\begin{array}{ll}
\texttt{element}(y, [x_1, \ldots, x_n], z_1'), & \texttt{element}(y, [c_{11}, \ldots, c_{1n}], w_1), \\
\texttt{element}(y, [d_1, \ldots, d_n] \times z_1', z_1), & \texttt{element}(y, [c_{21}, \ldots, c_{2n}], w_2), \\
\texttt{element}(y, [x_1, \ldots, x_n], z_2'), & \cdots \\
\texttt{element}(y, [d_1, \ldots, d_n] \times z_2', z_2), & \texttt{element}(y, [c_{m1}, \ldots, c_{mn}], w_m).
\end{array}$$

Note that $d_y x_y$ has to be replaced by a single variable in each constraint to avoid nonlinearity.

There are two basic optimizations to be made when compiling variable subscripts; *common subscript elimination* and *subscript folding*. The first consist simply of detecting if the same subscripts occurs more than once, and if so, reuse the `element` constraint generated. This can be done across the whole problem, i.e., the reuse need not be limited to be within the same linear inequality. The second reduces the number of `element` constraints introduced, by folding several variable subscripted constants into one. This amounts to summing all the constants within the same linear expression pairwise and then constraining a single artificial variable with a

mixed `element` constraint. Revisiting our previous example, the constraints can be more compactly compiled into

$$z + w = 42, \quad z \geq 0,$$
$$\texttt{element}(y, [x_1, \ldots, x_n], z'), \tag{5.2}$$
$$\texttt{element}(y, [d_1, \ldots, d_n] \times z', z), \tag{5.3}$$
$$\texttt{element}(y, [\sum_{i \in S} c_{i,1}, \ldots, \sum_{i \in S} c_{i,n}], w). \tag{5.4}$$

As illustrated in the example above, our MLLP modeling language and solver support three kinds of subscripted expressions in linear functions:

$$\texttt{element}(y, [a_1, \ldots, a_n], z) \quad \Leftrightarrow z = a_y, \tag{5.5}$$
$$\texttt{element}(y, [x_1, \ldots, x_n], z) \quad \Leftrightarrow z = x_y, \tag{5.6}$$
$$\texttt{element}(y, [a_1, \ldots, a_n] \times x, z) \Leftrightarrow z = a_y x. \tag{5.7}$$

The modeler compiles general expressions containing variable subscripts by summing and chaining together (5.5)–(5.7), and the solver can propagate bounds and create cuts for these structures. We observe that (a) any linear expression with variable subscripts can be decomposed by the modeler using these three forms, and, (b) they are simple enough to propagate efficiently [85].

Note that subscript folding is limited by this vocabulary, subscripted variables can, e.g., not be folded. Also, some information may be lost when decomposing an expression into several `element` constraints. For example, the minimum lower bound for z' in our example above, derived using bounds propagation on (5.2), might occur at a different index than the minimum of the d_i's in (5.3). This decoupling might thus give a weaker bound on z than by indexing directly on $d_y x_y$ in a single `element` constraint. There would not be any benefit, however, from having a new variant of the `element` constraint for $d_y x_w$ (except in the special case of $y \equiv w$ as in the example above) as $d_y x_w$ is naturally decoupled by the two different indexing variables and no information is lost by decomposing $z = d_y x_w$ to $z' = x_w$ and $z = d_y z'$.

More complex expressions can be handled by a more general form of the `element` constraint, instead of using decomposition. As we alluded to in Sec. 5.1, there is an intimate connection between variable subscripts in linear expressions (the mixed `element` constraint) and disjunctive programming. In fact, the three forms of the mixed `element` constraint above, eq. (5.5)–(5.7), are special cases of a general mixed global disjunctive constraint

$$\texttt{disjunctive}(y, [A^1 x \leq b^1, \ldots, A^n x \leq b^n]),$$

where y indexes among the linear systems in the second argument. Given the domain of y, what can be inferred in terms of bounds and cuts on the continuous variables x? This constraint has to produce cutting planes to identify the convex hull of the disjunction

$$\bigvee_{i \in D_y} \left(A^i x \leq b^i \right),$$

which is far more complex than the inference for the three frequently and naturally occurring structures (5.5)–(5.7), although it has been studied in the literature [6].

The propagation rules for the mixed `element` constraints are similar to the discrete case. Let the interval $[\min(x_i), \max(x_i)]$ be defined by the bounds on x_i. Then upon change of the domain of y, we can compute $\min = \{\min(x_i) \,|\, i \in D_y\}$ and $\max = \{\max(x_i) \,|\, i \in D_y\}$ and add new bounds $\min \leq z \leq \max$ to the LP. Reversely, the bounds of z can be used to prune D_y. More on propagation and relaxations for the `element` constraint can be found in [85].

It is interesting to make a comparison with variable subscripts in constraint programming at this point. The `element` constraint has been widely used in the CLP community for a long time, but only in two simple discrete forms, a_y and x_y (the latter only rarely). The existence of many other discrete global constraints could be the reason for the limited use of variable subscripts in CLP, and its limited vocabulary is partly explained by the fact that the primitive constraints (the easily handled constraints comprising the constraint store) are different in IP (LP) and CLP (CP). Arc- and bounds consistency on a CLP constraint store, containing only domain constraints ($x \in D$), allows the decomposition of variable subscripts (by the modeler) without loss of domain reduction power. This is not the case when the constraint store is composed of linear inequalities, as in our case, and a more expressive vocabulary is needed.

Variable Subscripts in Bounds Bounds on variables are traditionally required to be constants at compile time, given by the modeler to the solver, derived automatically or simply set to some smallest and largest possible value.

We extend this by allowing variable subscripts in bounds expressions, where the variable is declared. Declaratively, this amounts to posting the constraints

$$l_y \leq x \leq u_y \tag{5.8}$$

where l [u] is the lower [upper] bound vector and y a discrete variable. Procedurally, the variable subscripted bounds have two drawbacks. First, they are naïvely compiled into

$$z_1 \leq x, \quad x \leq z_2, \tag{5.9}$$
$$\texttt{element}(y, [l_1, \ldots, l_n], z_1), \tag{5.10}$$
$$\texttt{element}(y, [u_1, \ldots, u_n], z_2), \tag{5.11}$$

which introduces two new variables, two mixed `element` constraints and two linear inequalities. Secondly, and a more refined trap for our solver, the values of x, z_1 and z_2 in a continuous solution may be such that no value $i \in D_y$ satisfies (5.10)–(5.11), despite the fact that some value $i \in D_y$ together with x satisfies the original variable subscripted bounds (5.8). Methods aimed at finding such satisfying values (see Sec. 5.4.1) might then fail unnecessarily. This happens in practice and impedes the solver in completing the solution and causes extra branching in the search.

These two problems are avoided by compiling the bounds directly into

$$\texttt{element}_{\leq}(y, [l_1, \ldots, l_n], x), \tag{5.12}$$
$$\texttt{element}_{\geq}(y, [u_1, \ldots, u_n], x), \tag{5.13}$$

where (5.12) [(5.13)] only performs lower [upper] bound propagation. Since the artificial variables are eliminated, values for y satisfying (5.12)–(5.13) will be found correctly. The mixed `element` constraint described above is declaratively equivalent to $\texttt{element}_{\leq} \wedge \texttt{element}_{\geq}$, and is referred to as $\texttt{element}_{=}$ in ambiguous cases.

5.3.2 Semi-continuous Piecewise Linear Functions

Piecewise linear functions arise in a variety of problems. In this section we introduce two new ways of modeling and solving piecewise linear structures. The first is through the previously introduced variable subscripts and the second uses a specialized mixed global constraint.

A piecewise linear function consists of a set of line segments, usually with joint endpoints but possibly disjoint, which we refer to as semi-continuous. The segments constrain the v-axis (output variable, e.g., price) wrt. the u-axis (input variable, e.g., quantity), and also constrain the u-axis if the segments are disjoint. Segment i, denoted by the variable y, goes from point $(\underline{u}_i, \underline{v}_i)$ to point $(\bar{u}_i, \bar{v}_i)$. We use $v(u^*)$ to refer to the v-value of the function at u-value u^*.

Using variable subscripts, we can model the function as

$$v = \underline{v}_y + c_y(u \Leftrightarrow \underline{u}_y), \quad \underline{u}_y \leq u \leq \bar{u}_y, \quad \underline{v}_y \leq v \leq \bar{v}_y, \quad y \in \{1, \ldots, n\},$$

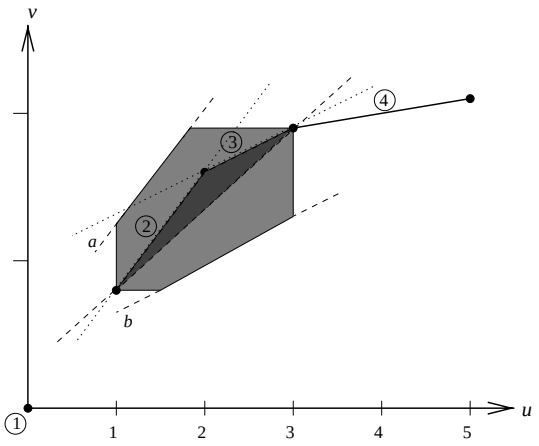


Figure 5.1: A piecewise linear function. The light grey area depicts the linear relaxation using variable subscripts, the dark grey area the relaxation obtained with the `piecewise` constraint.

where $c_i = (\bar{v}_i \Leftrightarrow \underline{v}_i)/(\bar{u}_i \Leftrightarrow \underline{u}_i)$ is the slope of segment i. As explained in the previous sections, this will be compiled into

$$v = z_1 + z_2 \tag{5.14}$$

$$\texttt{element}_{=}(y, [c_1, \ldots, c_n] \times u, z_1) \tag{5.15}$$

$$\texttt{element}_{=}(y, [\underline{v}_1 \Leftrightarrow c_1\underline{u}_1, \ldots, \underline{v}_n \Leftrightarrow c_n\underline{u}_n], z_2) \tag{5.16}$$

$$\texttt{element}_{\leq}(y, [\underline{u}_1, \ldots, \underline{u}_n], u) \tag{5.17}$$

$$\texttt{element}_{\geq}(y, [\bar{u}_1, \ldots, \bar{u}_n], u) \tag{5.18}$$

$$\texttt{element}_{\leq}(y, [\underline{v}_1, \ldots, \underline{v}_n], v) \tag{5.19}$$

$$\texttt{element}_{\geq}(y, [\bar{v}_1, \ldots, \bar{v}_n], v) \tag{5.20}$$

Consider the piecewise linear function in Fig. 5.1. It consists of four segments (numbers in circles), the first is the origin (of zero-width), the next three range between u-values 1–2, 2–3 and 3–5. At the depicted state the segment chosen is constrained to be 2 or 3, i.e., $y \in \{2, 3\}$. Bounds propagation in the `element` constraints will now produce the linear relaxation shown in light grey. The horizontal and vertical borders are obtained by updating the bounds for the continuous variables u and v, eq. (5.17)–(5.20). The other two borders (marked a and b) have slopes which are the effects of the bounds propagation of (5.15), and an offset which comes from (5.16). Note that the upper bound a is parallel to the steeper segment 2 and the lower bound b is parallel to segment 3.

Although the function is compactly described with variable subscripts, the solver does not know the true origin of the constraints, i.e., that it describes a piecewise linear function. As an alternative, consider introducing a new *mixed* global constraint representing a piecewise linear function,

$$\texttt{piecewise}(y, O, u, [\underline{u}_1, \ldots, \underline{u}_n], [\bar{u}_1, \ldots, \bar{u}_n], v, [\underline{v}_1, \ldots, \underline{v}_n], [\bar{v}_1, \ldots, \bar{v}_n]),$$

where y is the discrete variable indicating the segment in which the values of u and v lie, and O indicates the orientation of the constraint, i.e., whether v has to be above, below or on the piecewise linear function. Then a tighter linear relaxation can be produced by instead adding the cuts surrounding the dark grey triangle in Fig. 5.1. If O is above [below] only the lower [upper] cuts have to be posted, if it is on then both the lower and the upper cuts have to be posted. In addition to posting and updating the cuts described, the constraint performs bounds propagation on both the discrete and the continuous variables. For $D_y = \{p, \ldots, q\}$ the propagation rules for concave functions are:

$$\begin{array}{lcl} O \in \{\text{below, on}\} & : & v \leq \underline{v}_i + c_i(u \Leftrightarrow \underline{u}_i), \quad \forall i \in \{p, \ldots .q\}, \\ O \in \{\text{above, on}\} & : & v \geq (\bar{v}_q \Leftrightarrow \underline{v}_p)/(\bar{u}_q \Leftrightarrow \underline{u}_p)u + (\underline{v}_p \bar{u}_q \Leftrightarrow \bar{v}_q \underline{u}_p)/(\bar{u}_q \Leftrightarrow \underline{u}_p), \\ \text{Bounds on } u,v & : & \underline{u}_p \leq u \leq \bar{u}_q, \quad \underline{v}_p \leq v \leq \bar{v}_q. \end{array}$$

The second rule constrains the solution to lie above the line $(\underline{u}_p, \underline{v}_p) \Leftrightarrow (\bar{u}_q, \bar{v}_q)$. For convex functions the rules are swapped for above and below, and the inequalities are inverted.

These bounds and cuts provide the convex hull relaxation, which is the best we can do. Also, in the case of a concave above [below] piecewise linear function it is equivalent to the standard MIP [LP] relaxation (MIP and LP are reversed for convex functions), but introduces no new variables.

Declaratively, the `piecewise` constraint can be written in the form (5.1) as follows,

$$(y = i) \rightarrow \big(v = \underline{v}_i + c_i(u \Leftrightarrow \underline{u}_i),\ \underline{u}_i \leq u \leq \bar{u}_i,\ \underline{v}_i \leq v \leq \bar{v}_i\big),\ \forall i \in \{1, \ldots, n\},$$

but the `piecewise` extension of MLLP allows the use of the above described propagation rules which provides more inference.

The general syntax of the `piecewise` constraint, i.e., specifying both the start and end point of each segment, is necessary due to semi-continuous characteristic of a general piecewise linear function. Note, e.g., that a fixed-charge piecewise linear constraint is a special case of this constraint. Furthermore, the syntax allows for functions that are neither convex nor concave, and generating the convex hull for such general semi-continuous piecewise functions is then a matter of finding the convex hull of a set of points.

This is a well known problem in computational geometry and can be done efficiently [37]. A simpler syntax for continuous functions could of course be adopted, e.g., as in AMPL [59] and OPL [138], where the functions are assumed to be continuous and starting at the origin, and only the end point and slope of each segment are specified.

5.4 ALGORITHMIC EXTENSIONS

In CLP, good support for defining problem specific search strategies is essential. Strategies derive branching decisions from the current domains of variables and the structure of the constraint graph. Well-known strategies are *fail-first*, *most-constrained* and *earliest-start-time* (scheduling problems) for variable selection and *domain-splitting* for value choice [96].

In IP, the solution of the relaxation provides a complete labeling of the variables, i.e., each variable, whether continuous or integer, has a value in the solution to the relaxation. The benefit of this is twofold; (a) variables which require integral values might get integral values in the labeling "by chance" or might be roundable by a heuristic to an integral value, and, (b) if not, their fractional values provide valuable information for branching strategies.

The basic strategies from CLP remain available for a hybrid system like MLLP since the discrete decision variables still have domains and the discrete constraints still form a constraint graph. But in a solver where discrete and continuous variables have been separated, like in MLLP, discrete variables will not readily have values from the relaxation. The relaxation provides a value for x in (5.1) and a part of y might be determined by branching, but y is not given a value by the relaxation since it is not a part of it.

The relaxation and y are connected through conditional and mixed global constraints (`element`, `piecewise`, etc), and we need a new technique to communicate the impact of the linear relaxation solution to y. This section aims to show how this can be accomplished, and at the same time retain the ability of CLP to define custom search strategies.

5.4.1 Back-Propagation

At a node of the search tree, assume the following sequence of steps has been taken: (a) The branching choice has been enforced, (b) constraint propagation has been performed on discrete, logical and global constraints, and, (c) the resulting linear relaxation has been solved, with optimal solution x^*.

At this point, if all constraints are determined, we have a complete solution. If not, we might need to branch further. However, it might possible to

extend the solution x^* of the relaxation to a complete solution (y^*, x^*) that satisfies all constraints. And if not, we should deduce information for branching strategies.

Therefore, before branching, the following procedure, *back-propagation*, is performed. Once it is done, all the changes are undone before going on—the effects are local to this node. The procedure consists of executing a set of propagation rules, specific to each kind of constraint connecting the FD and LP store. The following are the rules for the constraints discussed in this paper, serving as examples:

- For any conditional constraint $h_i(y) \rightarrow A^i x \geq b^i$, enforce the constraint $\neg h_i(y)$ if $A^i x^* \not\geq b^i$.
- For any $\texttt{element}(y, (v_1, \ldots, v_k), z)$, where z occurs with a positive coefficient on the left-hand side-side of an (in)equality of type:

$$\begin{array}{rcl} \geq & : & \text{let } D_y = D_y \cap \{j \mid v_j^* \geq z^*\}, \\ \leq & : & \text{let } D_y = D_y \cap \{j \mid v_j^* \leq z^*\}, \\ = \text{ or both } \leq \text{ and } \geq & : & \text{let } D_y = D_y \cap \{j \mid v_j^* = z^*\}. \end{array}$$

- For any

$$\texttt{piecewise}(y, O, u, [\underline{u}_1, \ldots, \underline{u}_n], [\bar{u}_1, \ldots, \bar{u}_n], \\ v, [\underline{v}_1, \ldots, \underline{v}_n], [\bar{v}_1, \ldots, \bar{v}_n]),$$

  if:

$$\begin{array}{l} O \in \{\mathsf{below}, \mathsf{on}\} : \\ \quad \text{let } D_y = D_y \cap \{j \mid (\underline{u}_j \leq u^* \leq \bar{u}_j) \wedge (v^* \leq v(u^*))\}, \\ O \in \{\mathsf{above}, \mathsf{on}\} : \\ \quad \text{let } D_y = D_y \cap \{j \mid (\underline{u}_j \leq u^* \leq \bar{u}_j) \wedge (v^* \geq v(u^*))\}. \end{array}$$

  This will reduce D_k to (a) $\varnothing$, if u^* is outside a segment, (b) a singleton, if it is within a single segment, or, (c) a domain of cardinality two, if u^* is on the joint point of two segments.

After these rules have been applied, standard constraint propagation is performed for all purely discrete constraints. All elements of domains of discrete variables are now consistent with the current LP solution. If all constraints are determined, we have a complete solution. If some domain is empty, we say the back-propagation *failed* for this variable, and we need to branch further. If no domain is empty, but some purely discrete constraints are still not determined, we can search for a solution within the current domains

that determines those constraints. If we choose not to, or we fail to find one, we continue to branch.

So far we have achieved our equivalent of item (a) above, i.e., we take advantage of the fact that some discrete variables will "be integral in the labeling by chance", or as we would put it, will "have discrete values satisfying the continuous constraints by chance".

We refer to this technique as back-propagation for two reasons; (a) it is a form of inference, specifically tailored for each constraint, and (b) the constraints communicate their inferences back from the relaxation to the domains of shared variables. It should be pointed out that unlike the standard propagation, mixed constraints only back-propagate once, i.e., the back-propagation need not be encapsulated in a fix-point iteration. Recall also that since back-propagation is inference from *one* solution of *many* possible to the relaxation, its effect are node-specific, and must be undone before branching further.

In [50] a similar scheme called *unimodular probing*, is used to derive constraint violations from a totally unimodular subset of linear inequalities solved as an LP. The violated constraints indicates possible branching choices. Similarly, the value extraction for *shadowed* variables, i.e., a variable present in both CP and LP [127, 138], is also a special case of back-propagation, allowing branching strategies based on the LP solution values.

5.4.2 Branching Strategies

In a branching strategy there are two choices to make; what *variable* to branch and on, and what *value* to set it to. In IP it is very common to branch on the most fractional variable, resolving the "biggest" inconsistencies first. Using the domains after back-propagation, we can accomplish something similar, called *back-propagation-failure*. This strategy selects first any variable whose domain became empty during back-propagation, and secondly any variable whose domain after back-propagation was not singleton, and finally falls back on a standard fail-first strategy on the domains as they were before back-propagation. Variables with singleton domains after back-propagation are skipped since they have a consistent value.

For the value choice, a common choice in IP is to create two branches, $x \leq \lfloor x^* \rfloor$ and $x \geq \lceil x^* \rceil$, and branch on the one closer to integral first. The equivalent in MLLP is to first chose a value y^* which is consistent with one, several or all of the continuous constraints. We can choose to do domain splitting on this value (called *split-on-lp*), creating branches $y \leq y^*$ and $y \geq y^* + 1$. We can also, which makes even more sense, create three branches (called *triple-on-lp*): $y = y^*$, $y \leq y^* \Leftrightarrow 1$ and $y \geq y^* + 1$, preferably tried in that order.

All of the above value heuristics can also be combined with a *best-branch* selection, i.e., the objective value for the branches are computed, and the branches are tried in the order of greatest potential (best objective value). Computing the objective value before choosing a branch should not be confused with *strong-branching* [38]. Strong branching precomputes objective values for use in *variable* selection and will therefore incur overhead by solving relaxations for branches never taken. In contrast, *best-branch* is a *best-bound* node selection strategy, where the set of candidate nodes are the children of the current node. It will only move some computation up in the search tree, since all branches will eventually be considered, and thus not cause any overhead.

5.5 A PRODUCTION PLANNING PROBLEM

Consider a plant where a number of resources are used to manufacture a set of products, each produced unit requiring a certain amount of each resource. The objective is to maximize the profit. The revenue for a product does not increase proportionally with volume, since larger volumes gives discount to the buyer of the products, and is therefore approximated using a piecewise linear function. Similarly, resources are also volume discounted, and in addition, there is a minimum buying quantity for each resource if it is bought, which makes the corresponding piecewise linear function semi-continuous, i.e., a fixed-charge piecewise linear function. There are two additional constraints on production. First, there is a limit (plant capacity) on the total production. Second, the production equipment requires each product to be produced in a certain scale, e.g., small-scale, medium-scale, large-scale, etc., which limits the production to corresponding disjoint intervals.

5.5.1 An MLLP Model

Using the piecewise global constraint described in Sec. 5.3.2, the formulation is straightforward. Variable r_j is the revenue for product j, c_i the cost of resource i, y_j^R the segment where production v_j lies for product j, y_i^C the segment where usage u_i lies for resource i, and s_j is the production scale for product j. Parameters $\underline{u}_{ik}$ and $\bar{u}_{ik}$ denote upper and lower range of segment k for resource i, and $\underline{C}_{ik}$ and $\bar{C}_{ik}$ the corresponding accumulated cost at those points. Similarly, $\underline{v}_{jk}, \bar{v}_{jk}$ and $\underline{R}_{jk}, \bar{R}_{jk}$ describe the piecewise linear revenue function. Note that $\underline{u}_{i1} = \bar{u}_{i1} = 0$ and $\underline{u}_{i2} > 0$, enforcing

the minimum purchase quantity directly through the global semi-continuous piecewise constraint.

$$\max \sum_j r_j \Leftrightarrow \sum_i c_i$$

$$\text{s.t. } \texttt{piecewise}(y_i^C, u_i, \underline{u}_{i1}, \ldots, \underline{u}_{in}, \bar{u}_{i1}, \ldots, \bar{u}_{in}, c_i, \underline{C}_{i1}, \ldots, \underline{C}_{in}, \bar{C}_{i1}, \ldots, \bar{C}_{in}), \quad \forall i, \quad (5.21)$$

$$\texttt{piecewise}(y_j^R, v_j, \underline{v}_{j1}, \ldots, \underline{v}_{jm}, \bar{v}_{j1}, \ldots, \bar{v}_{jm}, r_j, \underline{R}_{j1}, \ldots, \underline{R}_{jm}, \bar{R}_{j1}, \ldots, \bar{R}_{jm}), \quad \forall j, \quad (5.22)$$

$$\sum_j a_{ij} v_j \leq u_i, \quad \forall i, \quad (5.23)$$

$$\sum_j v_j \leq \mathit{plant\_cap}, \quad (5.24)$$

$$\underline{w}_{j,s_j} \leq v_j \leq \bar{w}_{j,s_j}, \quad \forall j, \quad (5.25)$$

$$u_i, c_i, v_j, r_j \geq 0, \quad \forall i, j,$$

$$y_i^C \in \{1, \ldots, n\},\ y_j^R \in \{1, \ldots, m\},\ s_j \in \{1, \ldots, l\}, \quad \forall i, j.$$

Note the variable subscripted bounds (5.25) enforcing the restrictions on the scale of production.

5.5.2 Other Models

An IP Model: A traditional IP model, see Fig. 5.2, can be found by expanding the `piecewise` constraints to LP constraints for the revenue (concave maximization (`below`), constraints (5.26)–(5.27)), where $\hat{R}_{jk}$ is the revenue of each unit produced in interval k, and MIP constraints for the cost (concave minimization (`above`), constraints (5.28)–(5.31)). Due to lack of variable subscripts, the production scale intervals have to be encoded as a MIP structure (constraints (5.32)–(5.34)).

Notice that y_{jk} are SOS–3 variables [38] due to (5.34), which can be used by the IP solver. Informing the IP solver about the SOS–3 variables is optional, the model is still valid without that information. It is necessary, however, to tell the solver about the SOS–2 variables, λ_{ik}, as the model is incorrect without that information.

Finally, note that the semi-continuous minimum purchase quantity is enforced through an extra constraint (5.31).

An OPL Model: The modeling language OPL [138], like MLLP, allows formulations which mix discrete and continuous constraints. Variable subscripts are supported, but only for discrete variables or constants and not

$$\max \quad \sum_{j,k} \hat{R}_{jk} v_{jk} \Leftrightarrow \sum_{i} \left(\underline{C}_{i2} \lambda_{i1} + \sum_{k \geq 2} \bar{C}_{ik} \lambda_{ik} \right)$$

$$\text{s.t.} \quad v_j = \sum_k v_{jk}, \qquad \forall j, \tag{5.26}$$

$$v_{jk} \leq \bar{v}_{jk} \Leftrightarrow \underline{v}_{jk}, \qquad \forall j,k, \tag{5.27}$$

$$u_i = \underline{u}_{i2} \lambda_{i1} + \sum_{k \geq 2} \bar{u}_{ik} \lambda_{ik}, \qquad \forall i, \tag{5.28}$$

$$\sum_k \lambda_{ik} = 1, \qquad \forall i, \tag{5.29}$$

$$\{\lambda_{i1}, \ldots, \lambda_{in}\} \text{ is an SOS–2 set}, \qquad \forall i, \tag{5.30}$$

$$\underline{u}_{i2} b_i \leq u_i \leq M b_i, \qquad \forall i, \tag{5.31}$$

$$v_j = \sum_k w_{jk}, \qquad \forall j, \tag{5.32}$$

$$\underline{w}_{jk} y_{jk} \leq w_{jk} \leq \bar{w}_{jk} y_{jk}, \qquad \forall j,k, \tag{5.33}$$

$$\sum_k y_{jk} = 1, \qquad \forall j, \tag{5.34}$$

$$\sum_j v_j \leq \mathit{plant\_cap}, \tag{5.35}$$

$$\sum_j a_{ij} v_j \leq u_i, \qquad \forall i, \tag{5.36}$$

$$v_j, v_{jk}, w_{jk}, u_i, \lambda_{ik} \geq 0, \qquad \forall i,j,k,$$

$$y_{jk} \in \{0,1\}, \qquad \forall i,j.$$

Figure 5.2: An IP model for the production planning problem.

in continuous linear constraints nor in bounds. There is also a construct for modeling piecewise linear functions, but this construct is merely syntactic sugar for an LP or MIP structure, resolved at compile time (same as in AMPL [59]). Furthermore, the piecewise construct does not allow semi-continuous functions.

Nonetheless, the problem can be stated in a fairly compact manner in OPL, see Fig. 5.3. The variable subscripts in the bounds, enforcing the restrictions on the scale of the production, are set on integer variables v'_j which are shadow copies of the variables v_j in the LP (constraints (5.39)–(5.40)).

Unless specified, OPL applies a default branching strategy, which we assume varies with the problem, but whether it is in fact so is not documented in

$$
\begin{aligned}
\max\ & \left(\sum_j \texttt{piecewise}\{\hat{R}_{jk} \to \bar{v}_{jk};\ 0\}\ v_j\right) \Leftrightarrow \\
& \left(\sum_i \texttt{piecewise}\{\hat{C}_{ik} \to \bar{u}_{ik};\ M\}\ u_i\right) \\
\text{s.t. } & \underline{u}_{i2} b_i \le u_i \le M b_i, && \forall i, && (5.37)\\
& \sum_j a_{ij} v_j \le u_i, && \forall i, && (5.38)\\
& \underline{w}_{j,s_j} \le v'_j \le \bar{w}_{j,s_j}, && \forall j, && (5.39)\\
& v_j = v'_j, && \forall j, && (5.40)\\
& \sum_j v_j \le \mathit{plant\_cap}, && && (5.41)\\
& v_j, u_i \ge 0, && \forall i, j,\\
& v'_j \in \{0, \ldots, \infty\}, && \forall j,\\
& s_j \in \{1, \ldots, l\}, && \forall j,\\
& b_i \in \{0, 1\}, && \forall i.
\end{aligned}
$$

Figure 5.3: An OPL model for the production planning problem.

the OPL manual [138]. We tried the following strategy, trying to to some extent simulate the effect of back-propagation of MLLP:

search {
 forall (j in Products)
  tryall (k in ProdScales ordered by increasing
  abs(simplexValue($v[j]$) $\Leftrightarrow wl[j,k]$))
   $s[j] = k$;
};

However, benchmarking (Sec. 5.5.3) showed that the default strategy consistently performed better than what we could achieve with this strategy.

Note that the OPL is data dependant in the same way as the IP model. Both assume that the piecewise linear functions are continuous and thus the minimum purchase quantity has to be modeled specifically. In the OPL model it is enforced through an extra constraint (5.37).

Problem	**MLLP Model**						**Solution**		
Prod×Res	FD-Var	Elem	Piece	Row	Col	NZ	**Nodes**	**Opt**	**Time**
5×5	10	10	5	19	41	148	33	14	0.37
10×5	15	20	5	24	71	263	27	13	0.38
10×10	20	20	10	34	81	343	55	19	0.77
$5 \times 10^{*}$	15	10	10	29	51	203	127	23	1.00
$10 \times 15^{*}$	25	20	15	44	91	423	6874	33	50.84
$15 \times 15^{**}$	30	30	15	49	121	420	6044	63	52.46
$7 \times 12^{***}$	19	14	12	35	67	155	50	23	0.83

Table 5.1: Benchmark results for the MLLP model.

5.5.3 Benchmarks

In this section we evaluate the performance of three solvers on these three models. We used the same data sets for the different models; each number shown in the tables is the average for a model over 10 randomly generated data sets. The instances marked by a star are made harder (in addition to increased size) by forcing the resource usage close to the minimum purchase quantity. The unmarked and star-marked instances are *dense* in the sense a product requires some of all of the resources, i.e., the table indicating how much of each resource every product needs has no zero entries. The double star-marked instance has medium density and the triple star-marked instance is very sparse.

Superscripted digits indicates the number (out of 10 problems) that were solved to proven optimality within 100000 nodes (if at least one was solved), '–' indicates that none of the ten instances could be solved. 'Nodes' is the number of nodes required to prove optimality, and 'Opt' is the node in which the optimal solution was found, on average. All numbers include the problems that were not solved to optimality, i.e., the 100000 nodes and the time[1] it took to process them are weighed in.

Table 5.1 shows the results obtained with our MLLP solver . We branch on the piecewise intervals and the production scales simultaneously, selecting variables according to *back-propagation-failure* and values using *triple-on-lp* (see Sec. 5.4.2 for details).

Table 5.2 shows the performance of CPLEX 6.0.1 with default settings on the IP model. 'Bool.' indicates the number of 0–1 variables in the problem, 'PP' stands for "after preprocessing", and 'NZ' indicates the number of nonzero coefficients in the LP matrix. CPLEX uses best-bound node selection by default, i.e., it does not perform a depth first search. For sake of comparison with MLLP and OPL, Tab. 5.3 shows benchmarks results for

[1] Sun Ultra 60 Model 2360 (2×360 MHz UltraSPARC-II) running Solaris 2.6.

Problem	**IP Model**						**CPLEX Default**		
Prod×Res	Row	Col	Bool	Row^{PP}	Col^{PP}	NZ^{PP}	**Nodes**	**Opt**	**Time**
5×5	104	137	35	82	115	330	120	111	0.08
10×5	179	227	65	147	195	570	267	259	0.23
10×10	204	272	70	162	230	710	655	595	0.59
$5 \times 10^{*}$	129	182	40	97	150	445	1319	1048	0.85
$10 \times 15^{*}$	229	317	75	177	265	850	4484	1303	4.26
$15 \times 15^{**}$	304	407	105	242	345	1093	11316	3649	12.70
$7 \times 12^{***}$	169	236	54	125	193	555	12563	5352	8.10

Table 5.2: IP model, default settings.

Problem	**CPLEX Comparative**		
Prod×Res	**Nodes**	**Opt**	**Time**
5×5	1375	1357	0.72
10×5	1778	1717	1.31
10×10	8987	8929	7.59
$5 \times 10^{*}$	38721	38416	19.60
$10 \times 15^{*}$	–	–	–
$15 \times 15^{**}$	–	–	–
$7 \times 12^{***}$	21948	17163	13.69

Table 5.3: IP model, depth-first search and maximum infeasibility variable selection.

CPLEX with depth-first search and maximum infeasibility variable selection.

It should be noted that we tested a couple of different IP models using the MIP solvers CPLEX 6.0.1, XPRESS-MP 11.04 and Super LINDO 5.3. We chose to use CPLEX and the IP model above in these benchmarks since that combination exhibited the consistently best behavior. It should be emphasized, however, that unlike the MLLP model, the IP and OPL models are highly data dependant and inflexible. Both models assume continuous data and the minimum purchase quantity thus has to be modeled separately; if the structure of the data changes then the IP and OPL models have to be modified but the MLLP model will remain the same. It is possible to formulate the IP to handle any kind of data using an SOS–3 type of formulation instead of a mix of SOS–2 and SOS–3 but it had a significantly worse performance on the data we used.

Table 5.4 shows the performance of OPL. The number of rows and columns is after the piecewise constructs have been expanded to LP and MIP variables and constraints.

Problem	**OPL Model**					**Solution**	
Prod×Res	FD-Var	Elem	Piece	Row	Col	**Nodes**	**Time**
5×5	15	10	10	73	227	10774^{9}	6.51
10×5	25	20	15	133	392	73115^{3}	101.52
10×10	30	20	20	193	502	72143^{3}	135.63
$5 \times 10^{*}$	20	10	15	106	310	21429^{8}	21.57
$10 \times 15^{*}$	35	20	25	238	597	67440^{4}	82.24
$15 \times 15^{**}$	45	30	30	301	765	80888^{2}	105.85
$7 \times 12^{***}$	26	14	19	97	363	80023^{2}	114.32

Table 5.4: OPL model, default settings

Comments on Computational Results Our solver performs well compared to both CPLEX (IP model) and OPL (OPL model). In particular, MLLP is much faster at finding the optimal solution. Several other things should be noted in the tables. First, the LPs solved by MLLP are both smaller and more compact than the corresponding LPs in the IP model. Nevertheless, MLLP spends more time per node. This is expected, our code is a research tool and not tuned for performance; except for the LP code, which uses the CPLEX callable libraries, the MLLP solver is coded in Java, including the branch-and-bound search and the propagation. Back-propagation currently takes a significant portion of the time (50–70% depending on problem), but we believe this can be remedied in a tuned implementation; the process is worst-case linear in the number of constraints and can be coded efficiently. Solving LPs accounts for 20–40%, including time for communication with CPLEX.

The OPL model is very similar to the MLLP model at a first glance, but the underlying differences contribute to the poor performance of OPL on these problems. First, the piecewise construct of OPL compiles to an LP model for the product revenue functions and a MIP model with boolean variables for the resource costs, which is probably quite similar to our IP model. OPL provides no back-propagation, and the possible search strategies are limited by the fact that the boolean variables of the piecewise cost function are not visible to the user, and can not be used to customize the search strategy. The default strategy is used, and judging from the fact that OPL performs slightly better for the instances marked by a star, which have less resource usage, OPL probably enumerates values for integer variables. A more adapted search strategy, if possible, would probably improve OPL's robustness and performance significantly.

5.6 CONCLUSION

Our focus in this paper is on modeling for and solving with a combined constraint propagation–linear programming solver. We show for a production planning problem that we can greatly reduce the LP relaxation size while retaining its strength, and take advantage of constraint propagation for inference from branching decisions and discrete constraints. Computational testing shows that our approach is competitive with commercial IP codes.

An important part of a hybrid modeling language are variable subscripts for the continuous domain, in which discrete variables are used to index into arrays of continuous constants or variables. We introduce new variants of the classical `element` constraint from constraint programming, and show how general subscripted expressions can be compiled and decomposed to such constraints for compact models and efficient problem solving.

We describe a scheme for inference from an LP solution to a discrete constraint store, which is an essential tool to avoid excessive branching. This inference allows the solver (which separates discrete and continuous variables) to earlier complete a feasible continuous solution to include values for discrete variables. It also provides the information necessary to make intelligent branching decisions, much equivalent to how IP uses fractional values for integer variables in branching strategies.

We also show how a mixed global constraint modeling a piecewise linear function can reduce the LP size while retaining an equivalent relaxation. Even more, it is also shown to increase the inferences made both from the FD store to the LP store (cutting planes) and from LP to FD (back-propagation of LP solution). Global constraints crossing the boundary between CLP and IP have great potential, mainly so for the same reasons as global constraints have shown to be indispensable in pure CLP. They allow a more compact representation, increase readability, and most importantly, improve inference. In addition, a flexible global constraint can be more robust to model and data changes, e.g., our piecewise constraint, which allows any kind of semi-continuous, concave/convex function as input without any modification to the model.

Possibly, mixed global constraints will be even more powerful than traditional global constraints, since a constraint store with (continuous) linear inequalities (the primitive constraints of LP) is more expressive than the finite `indomain` constraints (the primitive constraints of FD). The use of LP as a constraint store have not been fully explored in the CLP community, especially not as a store for communication between (global) constraints.

Linear Relaxations and Reduced-Cost Based Propagation of Continuous Variable Subscripts

Greger Ottosson and Erlendur S. Thorsteinsson

In hybrid solvers for combinatorial optimization, combining Constraint (Logic) Programming (CLP) and Mixed Integer Programming (MIP), it is important to have tight connections between the two domains. We extend and generalize previous work on automatic linearizations and propagation of symbolic CLP constraints that cross the boundary between CLP and MIP. We also present how reduced costs from the linear programming relaxation can be used for domain reduction on the CLP side. Computational results comparing our hybrid approach with pure CLP and MIP on a configuration problem show significant speed-ups.

6.1 INTRODUCTION

The topic of this paper can be seen as the merger of three lines of research in the area of hybrid techniques of Constraint (Logic) Programming (CLP) and Mixed Integer Programming (MIP).

Firstly, while studying a configuration problem, we continue along the lines of Rodošek, Wallace and Hajian [127] in providing automatic linearizations of arithmetic and symbolic CLP constraints. The object of study here are

terms with variable subscripts in continuous constraints, i.e., a structure which in part is modeled with the `element` constraint in CLP.

Secondly, we continue a line of research by Focacci, Lodi and Milano [52, 53, 54, 55], who have used the *reduced costs* of a separate assignment subproblem for propagation in a CLP framework. We generalize this by showing how reduced-cost based inference can be applied to constraints whose linearization is a part of a larger linear programming relaxation.

Finally, we show how this fits nicely into the modeling framework Mixed Logical/Linear Programming (MLLP) and its implementation as a hybrid CLP-MIP solver, which is a part of our previous line of research [84, 85, 105, 108].

This paper is structured as follows. Section 6.2 introduces our application, a class of configuration problems, with CLP, MIP and hybrid MLLP models. Sections 6.3 and 6.4 describe how the variable subscripts are linearized in MLLP. Section 6.5 describes reduced costs and how they can be used for inference using those linearizations. Finally, Section 6.6 gives computational results.

6.2 A CONFIGURATION PROBLEM

Many industrial products come in different configurations, aiming to closely satisfy the needs of individual customers. In one class of such problems there are components, each one of a set of possible types, and the aim is to find a feasible configuration with a type and quantity for each component that optimizes some criteria. Components supply or consume attributes/resources (such as weight, cost or effect), and these resources are constrained or are a part of the objective. In our problem all quantities are integral and in addition there are a set of logical side-constraints, the *configuration* constraints.

Given is a set of components C, possible component types T_i for component i, and a set of attributes R. Variable t_i is the type of component i, q_i is the quantity of component i and r_k is the quantity of attribute/resource k. The weight/cost per unit of attribute/resource k is denoted by c_k, R_k is the minimum amount of attribute/resource k used/produced and $A_{k,i,j}$

defines how many units of attribute/resource k are produced/consumed by each component i if it is of type j. Let $q = (q_i)_{i \in C}$ and $t = (t_i)_{i \in C}$. Then,

$$\begin{aligned}
\min \quad & \sum_{k \in R} c_k r_k \\
\text{s.t.} \quad & r_k = \sum_{i \in C} \Big(q_i \times A_{k,i,t_i} \Big), && \forall k \in R, && (6.1) \\
& h_l(q,t), && \forall l \in L, && (6.2) \\
& r_k \geq R_k, && \forall k \in R, && (6.3) \\
& t_i \in T_i,\ q_i \in \mathbb{Z}_+, && \forall i \in C.
\end{aligned}$$

Note that the resource consumption "lookup" is handled in (6.1) through a variable subscript on A, i.e., A_{k,i,t_i} where the *variable* t_i is the type of component i. This term is the core of the problem, and is, as we shall see, modeled differently in CLP and IP. Equation (6.2) states the set of configuration constraints, e.g., $q_1 > 0 \Rightarrow q_2 = 0$ or $\texttt{alldiff}(t_1, \ldots, t_n)$.

6.2.1 A CLP Model

In CLP the configuration problem is modeled with variable subscripts (`element` constraints) as follows (assuming that $T_i = \{1, \ldots, n_i\}$):

$$\begin{aligned}
\min \quad & cr \\
\text{s.t.} \quad & r_k = \sum_{i \in C} q_i a_{ki}, && \forall k \in R, && (6.4) \\
& \texttt{element}(t_i, [A_{ki1}, \ldots, A_{kin_i}], a_{ki}), && \forall i \in C, k \in R, && (6.5) \\
& r_k \geq R_k, && \forall k \in R, && (6.6) \\
& t_1 = 1 \Rightarrow t_2 \in \{1, 2\}, && && (6.7) \\
& q_1 > 0 \Rightarrow q_2 = 0, && && (6.8) \\
& t_i \in T_i,\ q_i \in \mathbb{Z}_+,\ a_{ki} \in \mathbb{Q}, && \forall i \in C, k \in R.
\end{aligned}$$

where a_{ki} is the consumption of resource k by component i, and the other variables and constants as before. Constraint (6.7)–(6.8) are two examples of a logical side-constraints. Note that (6.4) is non-linear and that there are $|C| \times |R|$ `element` constraints.

6.2.2 A MIP Model

In a MIP model, the lack of variable subscripts and the linear requirement means that the component types t_i have to be replaced with 0–1 variables t_{ij}, where t_{ij} is 1 if component i has type j, 0 otherwise. Similarly, the

quantity of the i-th component, q_i, is disaggregated into q_{ij} for types $j \in T_i$ (constraints (6.11)–(6.12)):

$$
\begin{aligned}
\min \quad & cr \\
\text{s.t.} \quad & r_k = \sum_{i \in C} \sum_{j \in T_i} q_{ij} A_{kij}, && \forall k \in R, && (6.9)\\
& \sum_{j \in T_i} t_{ij} = 1, && \forall i \in C, && (6.10)\\
& q_{ij} \leq M t_{ij}, && \forall i \in C, j \in T_i, && (6.11)\\
& q_i = \sum_{j \in T_i} q_{ij}, && \forall i \in C, && (6.12)\\
& r_k \geq R_k, && \forall k \in R, && (6.13)\\
& t_{21} + t_{22} \geq t_{11}, && && (6.14)\\
& b \leq q_1 \leq M b,\ q_2 \leq M(1 \Leftrightarrow b), && && (6.15)\\
& t_{ij} \in \{0,1\},\ q_i, q_{ij} \in \mathbb{Z}_+,\ b \in \{0,1\}, && \forall i \in C, j \in T_i. &&
\end{aligned}
$$

Equations (6.14)–(6.15) in this model formulate the configuration constraints from before.

6.2.3 An MLLP Model

The MLLP model is similar to the CLP model, although the underlying solution strategy employs LP relaxations in conjunction with the constraint propagation:

$$
\begin{aligned}
\min \quad & cr \\
\text{s.t.} \quad & r_k = \sum_{i \in C} q_i A_{kit_i}, && \forall k \in R, && (6.16)\\
& r_k \geq R_k, && \forall k \in R, && (6.17)\\
& t_1 \in \{1\} \Rightarrow t_2 \in \{1,2\}, && && (6.18)\\
& b \in \{0\} \Rightarrow q_1 = 0,\ b \in \{1\} \Rightarrow (q_1 \geq 1, q_2 = 0), && && (6.19)\\
& t_i \in T_i,\ q_i \in \mathbb{Z}_+,\ b \in \{0,1\}, && \forall i \in C. &&
\end{aligned}
$$

Note that the cost "lookup" is handled through a variable subscript on A, i.e., by A_{kit_i} where the variable t_i is the type of component i. The next two sections describe the linear relaxations used for these variable subscripts, and how the relaxations link the discrete and continuous parts of the MLLP model.

6.3 LINEAR RELAXATION OF C_Y

The term $q_i \times A_{k,i,t_i}$ is the core of the configuration problem. We will, however, begin with a simpler linearization, of a single subscripted constant

c_y. This would be the term which would occur in the MLLP model if it was limited to unit quantities, i.e., if $q_i \equiv 1$, and is equivalent to the traditional `element` constraint in CLP. A subscripted constant, c_y, occurring alone in a term is compiled as in the following example:

$$\begin{array}{l} 2x + c_y \leq 18, \\ c \in \{1.0, 4.5, 6.1\}, \end{array} \iff \begin{array}{l} 2x + z \leq 18, \\ \texttt{element}(y, [1.0, 4.5, 6.1], z). \end{array}$$

Bounds-consistency on the `element` constraint involves producing increasingly tighter bounds on z as the domain D_y of y is reduced. A straightforward linearization of this constraint is identical to how this structure is modeled in MIP, i.e., we introduce decision variables $b_1, \ldots, b_k$ for $D_y = \{1, \ldots, k\}$, where b_i is 1 if $y = i$ and 0 otherwise, and add

$$\begin{array}{rll} & 0 \leq b_i \leq 1, \ \forall i, & (6.20) \\ \texttt{element}(y, [c_1, \ldots, c_k], z) \iff & b_1 + \ldots + b_k = 1, & (6.21) \\ & z = c_1 b_1 + \ldots + c_k b_k. & (6.22) \end{array}$$

to the LP. This linear relaxation is no stronger than bounds-consistency on the `element` constraint (actually equivalent to it). Thus there is little incentive to use this relaxation unless these linear constraints connect in a beneficial way to some other linear constraints, or useful information (dual values, reduced costs, etc.) can be derived and used by including them.

6.4 LINEAR RELAXATION OF $C_Y X$

We now turn our attention back to the basic structure (6.1) of the configuration problem. The term $c_y x$ is a subscripted constant multiplied by a continuous variable. In the configuration problem it denotes the cost of buying x units of type y at cost c_y each. In MLLP this structure is replaced by a continuous variable z when the model is compiled and the constraint $\texttt{element}(y, [c_1, \ldots, c_k] \times x, z)$ is introduced into the problem. For example

$$\begin{array}{l} 2x + c_y x \leq 18, \\ c \in \{1.0, 4.5, 6.1\}, \end{array} \iff \begin{array}{l} 2x + z \leq 18, \\ \texttt{element}(y, [1.0, 4.5, 6.1] \times x, z). \end{array}$$

When propagating this variant of the `element` constraint upon a change of D_y, the domain of y, cuts of the form

$$c_{\min} x \leq z \qquad \text{and} \qquad c_{\max} x \geq z \tag{6.23}$$

are updated, where $c_{\min} = \min\{c_i : i \in D_y\}$ and $c_{\max} = \max\{c_i : i \in D_y\}$.

This is compact, but a stronger relaxation of $\texttt{element}(y, [c_1, \ldots, c_k] \times x, z)$ is achieved by disaggregating x into bins x_i, and linking to the corresponding decision variable z:

$$0 \le b_i \le 1, \qquad \forall i, \quad (6.24)$$

$$b_1 + \ldots + b_k = 1, \quad (6.25)$$

$$x = x_1 + \ldots + x_k, \quad (6.26)$$

$$z = c_1 x_1 + \ldots + c_k x_k, \quad (6.27)$$

$$x_i \ge 0, \qquad \forall i, \quad (6.28)$$

$$x_i \le M b_i, \qquad \forall i. \quad (6.29)$$

Equations (6.24)–(6.25) are the same as for c_y above, but in addition, x is disaggregated to $x_1, \ldots, x_k$, and connected through the big-M constraints (6.29) to $b_1, \ldots, b_k$. Intuitively, this relaxation is stronger than bounds-consistency simply because all the coefficients are weighed into the LP relaxation. As we shall see, this makes a big difference in computational efficiency.

This formulation is not the smallest convex hull relaxation possible. The purpose of the variables $b_1, \ldots, b_k$ and the big-M constraints (6.29) is only to ensure that at most one of $x_1, \ldots, x_k$ is non-zero. This is unnecessary in our framework since this is implicitly enforced by the connection to y. As an alternative to this intuitive explanation, consider deriving the convex hull formulation of this constraint from the general disjunctive formulation [6]. The disjunction equivalent of $\texttt{element}(y, [c_1, \ldots, c_k] \times x, z)$ is

$$\bigvee_{i \in D_y} \big(\Leftrightarrow z + c_i x = 0 \big), \quad (6.30)$$

and the general disjunctive formulation a) applied to (6.30) gives b) since $\alpha = 0$, which then simplifies to c), which are equations (6.26)–(6.27) above.

$$\begin{array}{lll}
\bigvee a_i x \le \alpha, & & \\
\quad \Updownarrow & (z, x) = \sum_i (z_i, x_i), & \\
x = \sum_i x_i, & \Leftrightarrow z_i + c_i x_i \le 0, \quad \forall i, & z = \sum_i c_i x_i, \\
a_i x_i \le \alpha b_i, \quad \forall i, & z_i \Leftrightarrow c_i x_i \le 0, \quad \forall i, & \\
\sum_i b_i = 1. & \sum_i b_i = 1. & x = \sum_i x_i. \\
\text{a)} & \text{b)} & \text{c)}
\end{array}$$

Thus, we can dispose of (6.24), (6.25) and (6.29), given that upon domain reduction $i \notin D_y$ of y, we enforce $x_i = 0$. This is similar to the modeling and branching with Special Ordered Sets (SOS) of type 1 in MIP.

This tight relaxation, which computational tests (Sec. 6.6) show is an invaluable tool, links the discrete and continuous parts of the MLLP model (Sec. 6.2.3), The continuous (linear) part includes the resource constraints

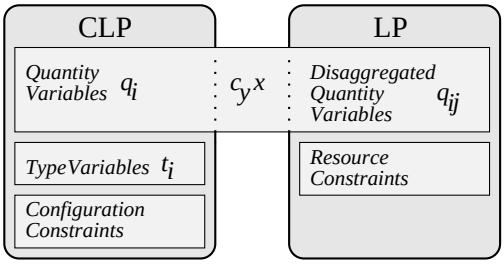


Figure 6.1: The build-up of the MLLP model

(6.16), with $q_i A_{kit_i}$ linearized as detailed above.[1] Figure 6.1 illustrates the effect of this. The quantities are present in both the CLP and LP parts, although in different forms. CLP has q_i, whereas the linearization introduces the q_{ij} variables into the LP. The component types are only present in the CLP part, because as proven above, they add nothing to the relaxation and are therefore not included.

6.5 REDUCED COSTS

Using reduced costs for inference in optimization methods is standard practice in OR and is available in several MIP solvers, e.g., CPLEX [86] and X-PRESS MP [43]. We will only give a brief description of reduced costs here, for more details see [35, 101]. In a basic solution to the Linear Program (LP),

$$\begin{aligned} \min \quad & z = cx \\ \text{s.t.} \quad & Ax = b,\ x \geq 0, \end{aligned}$$

the variables are partioned into *basic* and *non-basic* variables [35], call them x_B and x_N, respectively. If we partition the constraint matrix $A = [B\ N]$ and the objective function $c = [c_B\ c_N]$ in the same manner, we can write the problem above as

$$\begin{aligned} \min \quad & z = c_B x_B + c_N x_N \\ \text{s.t.} \quad & Bx_B + Nx_N = b, \\ & x_B, x_N \geq 0. \end{aligned} \tag{6.31}$$

[1] Unify q_i with x, t_i with y and A_{kit_i} with c, to map between model and linearization.

The solution to equation (6.31) is $x_B = B^{-1}b \Leftrightarrow B^{-1}Nx_N$. In a *basic solution*, x_N will be 0 and therefore $x_B = B^{-1}b$. We note that the value of the objective function will then be

$$\begin{aligned} z &= c_B x_B + c_N x_N = c_B(B^{-1}b \Leftrightarrow B^{-1}Nx_N) + c_N x_N \\ &= c_B B^{-1}b + \underbrace{(c_N \Leftrightarrow B^{-1}N)}_{\bar{c}} x_N = c_B B^{-1}b. \end{aligned}$$

The reduced costs, $\bar{c}$, are defined for the non-basic variables $x_N = \{x_{i_1}, \ldots, x_{i_k}\}$ (which are all zero in the basic solution). The reduced cost $\bar{c}_{i_j}$ then corresponds to the *cost per unit increase* to the objective function value if x_{i_j} would take on a non-zero value. The basic solution is an *optimal basic solution* if all the reduced costs are non-negative, indicating that there is no benefit in having any of the non-basic variables taking on non-zero values.

A process called *reduced cost fixing* [146] uses the reduced costs of relaxed 0–1 variables in MIP problems, obtained from the optimal linear programming relaxation solution $\bar{x}$, to potentially fix variables to zero without having to branch on them. This can be done for a non-basic variable x_{i_j} if $c\bar{x} + \bar{c}_{i_j} > cx^*$, i.e., if the current objective value plus the reduced cost $\bar{c}_{i_j}$ of the variable x_{i_j} exceeds the objective value of the incumbent solution x^*, the best solution found so far in the branch-and-bound search. This generalizes to variables at their lower *or* upper bound, and to general integer variables.

6.5.1 Reduced-Cost Based Propagation in CLP

Recently, Focacci et. al. [55, 52, 54] have adapted variable fixing to a CLP framework and used it successfully for the domain pruning of the `alldiff` constraint and the `path` constraint in ILOG Solver. This is done by shadowing these global constraints with a linear formulation of the assignment problem [146] and solving it to optimality while computing the corresponding reduced costs. This is done incrementally using the Hungarian algorithm, but it could also be done with linear programming. In the assignment problem, the variable $x_{ij} = 1$ corresponds to $x_i = j$ in $\texttt{alldiff}(x_1, \ldots, x_k)$. As before, if $c\bar{x} + \bar{c}_{ij} > cx^*$ for a non-basic variable x_{ij} and incumbent solution x^*, then $x_i \neq j$. This rule is used within a standard fix-point propagation loop and can thus interact and communicate with other constraints. From a CLP perspective, this is a new way of effectively using the objective function for domain reduction in specific global constraints/structures. On the other hand, from an OR point-of-view, reduced-cost fixing is given added value through its integration with other inference algorithms.

Note that the assignment problem is completely separate there from the rest of the model; it is only used in the propagation loop. The assignment

problem is also *totally unimodular* [101], which means that the optimal solution to the linear relaxation will always be integral. This will not be the case for general hybrid CLP–MIP models, but the reduced costs will still be well-defined for non-basic variables.

6.5.2 Reduced-Cost Based Propagation in MLLP

In a hybrid CLP–MIP solver, such as MLLP, where the inference of CLP is combined with a linear relaxation, we can also use reduced-cost based propagation, given that our constraints are linearized. The reduced cost propagation then becomes a part of the regular fixed-point propagation loop, which all the discrete and mixed constraints of MLLP are a part of.

Again we start with the simple subscripted constant, c_y (Sect. 6.3). Assume the current optimal LP solution is $\bar{x}$, with reduced costs $\bar{c}_1, \ldots, \bar{c}_k$ for the decision variables $b_1, \ldots, b_k$, and x^* is the incumbent MLLP solution. Then the following rule defines reduced-cost based domain reduction for the `element` constraint (assume minimization), if linearized by (6.20)–(6.22):

$$\begin{array}{l}\texttt{do } \forall i \in D_y \\ \quad \texttt{if } c\bar{x} + \bar{c}_i \geq cx^* \texttt{ then } D_y = D_y \Leftrightarrow \{i\}\end{array}$$

This domain pruning is equivalent to standard constraint propagation, i.e., valid in this subproblem and all its extensions (this node and below in the search tree). As we noted earlier, remember that the linearization of c_y is no stronger than simple bounds-consistency, but that it allows us to do this reduced-cost based propagation, which may be beneficial in some cases.

Returning to our configuration problem, a similar reduced-cost based propagation rule for the term $c_y x$ (Sect. 6.4) can be used on the linearization of (6.26)–(6.28).

$$\begin{array}{l}\texttt{do } \forall i \in D_y \\ \quad \texttt{if } c\bar{x} + \bar{d}_i \geq cx^* \texttt{ then } D_y = D_y \Leftrightarrow \{i\} \\ \texttt{do } \forall i \in D_y \\ \quad x \leq \lfloor (cx^* \Leftrightarrow c\bar{x})/\bar{d}_i \rfloor\end{array}$$

where $\bar{d}_1, \ldots, \bar{d}_k$ are the reduced costs for $x_1, \ldots, x_k$, respectively. The last line bounds the quantity variable; the standard variable fixing in MIP, which is also applicable in MLLP.

6.6 COMPUTATIONAL TESTING

The first interesting object of study is to compare how CLP, MIP and MLLP perform on this problem. Figure 6.2 shows average number of nodes and

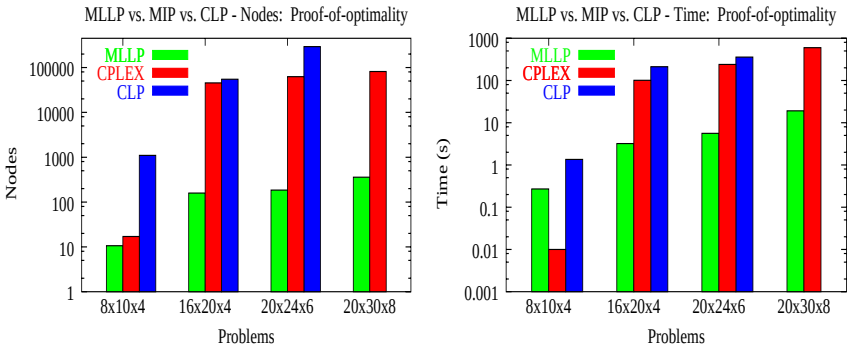


Figure 6.2: MLLP vs. MIP vs. CLP on a configuration problem

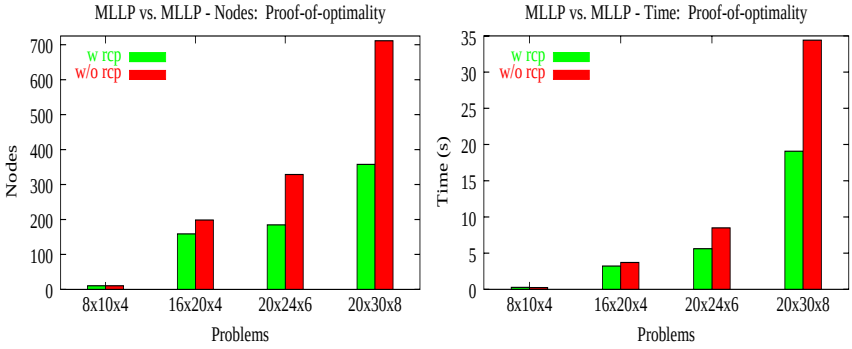


Figure 6.3: The impact of reduced-cost propagation, with RCP vs. without RCP

average time required to solve to proven optimality instances of different sizes (note the logarithmic scale). The first, easily solved, class is a set of seven instances with real-world data; the following classes have ten instances each, with data generated to closely resemble the original instances.

For size '16x20' (16 components, 20 types), CPLEX solves 8 of the 10 instances to proven optimality within 100 000 nodes, at which point the search is aborted. For size '20x24' it only solves 4 of the 10 instances and for size '26x30' it only solves 3 of the 10 instances. SICStus Prolog [51] solves 6 instances of size '16x20' to proven optimality within 500s (50 000–100 000 nodes), 7 instances of size '20x24' on and no '26x30' instances. MLLP solves all instances of all sizes, a magnitude or more faster. It should be emphasized that MLLP uses the relaxation of $c_y x$ described in Sec. 6.4, without which MLLP performs much worse.

The second method we want to study is the reduced-cost based propagation. It has been shown to be clearly beneficial in a CLP context [52, 54, 55], where it adds optimization oriented domain reduction, but is it equally effective in a framework like MLLP that is already focusing the search around the relaxation optimum? Figure 6.3 shows the effect on the number of nodes and the time required to solve to proven optimality. The reduction in nodes is significant, around 10-50% is saved (higher percentage as the problems grow larger), and the overhead is not considerable so the CPU time saved is almost as much.

6.7 CONCLUSION

In this paper, we presented and compared CLP, MIP and hybrid MLLP models for a configuration problem. For the hybrid MLLP model, we showed how a continuous variant of the `element` constraint, which is central to the problem, can be linearized and how the reduced costs from the LP relaxation can be used for domain pruning. Computational results were included, comparing different models and techniques on a set of real-world instances. The results showed that the linear relaxation introduced is an invaluable tool and that the reduced cost propagation also contributes to an efficient solution. The hybrid MLLP approach was shown to out-perform both pure CLP and MIP, and was able to solve instances larger than the other two methods could handle.

6.8 ACKNOWLEDGEMENTS

We would like to thanks Tacton Systems [103, 134] for providing the configuration problem and initial data, Mats Carlsson, Swedish Institute of Computer Science (SICS), for reading drafts of this paper, and Prof. John Hooker, Carnegie Mellon University (CMU), for general guidance and inspiration.

An MLLP Model for Production Planning

A.1 THE MODEL

```
#
# File: Production Planning
# Authors: Hak-Jin Kim,
#          Greger Ottosson,
#          Erlendur S. Thorsteinsson
#
# Description: MLLP model of a production planning
#              problem which has piecewise linear
#              profit and cost functions.

################### MODEL ##################

set Resources;  # Rescources needed to produce the...
set Products;   # ...output products.
set CostItv;    # Intervals for resources levels
set PriceItv;   # Intervals for product levels
set Scale;      # Scales of production (none, small
                # or large, etc)

### RESOURCES
#
# The number of consumed resources of each type
var Use {i in Resources};

# Resource level, i.e. in which interval the
# comsumption lies for each resource
var UseItv {i in Resources} integer, := CostItv;
```

```
# The cost for resources of each type (intermediate
# variable used primarily in objective).
var ResCost {Resources};


### PRODUCTS
#
# The number of produced products of each type.  Note
# the variable subscripts in the bounds.
var Prod {j in Products}
        >= wl[j, ProdScale[j]],
        <= wu[j, ProdScale[j]];

# Production scale, i.e. which production scale each
# product is set up for
var ProdScale {j in Products} integer, := Scale;

# The number of produced products in each production
# interval
var ProdVol {j in Products, k in PriceItv}
         <= (vu[j, k] - vl[j, k]);

### INPUT PARAMETERS
#
# Units resources required per product unit
param a {Resources, Products};

# Marginal costs for resources
param c {Resources, CostItv};

# Marginal prices for outputs
param r {Products, PriceItv};

# Lower level of resources in interval
param uu {Resources, CostItv};

# Upper level of resources in interval
param ul {Resources, CostItv};

# Lower level of products in interval
param vu {Products, PriceItv};

# Upper level of products in interval
param vl {Products, PriceItv};
```

```
# Large scale production of resources
param wu {Products, Scale};

# Small scale production of resources
param wl {Products, Scale};

# Maximum total production
param plant_capacity;

### PRECOMPUTED PARAMETERS:
#
# Cost for each resource of using interval 1..i
# to it's lower capacity
param dl {Resources, CostItv};

# Cost for each resource of using interval 1..i
# to it's upper capacity
param du {Resources, CostItv};


######### OBJECTIVE FUNCTION ##########

# The revenue to maximize is the sum of products
# produced times their price, minus the cost of all
# resources required.
var Z;
maximize profit: Z;

subject to objective: Z =
        sum {j in Products, k in PriceItv}
            r[j, k]*ProdVol[j, k] -
        sum {i in Resources} ResCost[i],
        Z >= 0;

######### CONSTRAINTS ##########

### PRODUCTION
#
# The sum of all production in all revenue
# intervals is the total production of each
# product.
subject to production_revenue: forall {j in Products}
        Prod[j] = sum {k in PriceItv} ProdVol[j, k];
```

```
# Maximum plant production capacity
subject to plant_capacity_cons:
        sum {j in Products} Prod[j] <= plant_capacity;


### RESOURCES
#
# Piecewise linear constraint
subject to resource_pwl: forall {i in Resources}
        piecewise UseItv[i], concave, minimize,
                  Use[i],     {k in CostItv} ul[i,k],
                              {k in CostItv} uu[i,k],
                  ResCost[i], {k in CostItv} dl[i,k],
                              {k in CostItv} du[i,k];


### LINK BETWEEN PRODUCTS AND RESCOURCES
#
# Link production with required resources.
subject to link: forall {i in Resources}
        Use[i] >= sum {j in Products} a[i, j]*Prod[j];


######### SEARCH ##########

var ToUse {i in Resources} boolean;

subject to Usage: forall {i in Resources}
        ToUse[i] in {1} -> UseItv[i] in {2..100};

subject to NoUsage: forall {i in Resources}
        ToUse[i] in {0} -> UseItv[i] in {1};

search {
        generate ordered by backfail
                 value by best enum
                {i in Resources} ToUse[i];
        generate ordered by backfail
                 value by best triple_on_lp
                {i in Resources} UseItv[i]
                {j in Products} ProdScale[j];
};
```

```
######### DATA ##########

param dl {i in Resources, l in CostItv} :=
        sum {k in {1..(l-1)}}
            c[i, k]*(ul[i, k+1]-ul[i, k]);

param du {i in Resources, l in CostItv} :=
    (sum {k in {1..(l-1)}}
         c[i, k]*(ul[i, k+1]-ul[i, k])) +
         c[i, l]*(uu[i, l]-ul[i, l]);
```

A.2 A DATA INSTANCE

```
######### PRODUCTION PLANNING DATA ##########

set Resources := {1..3};
set CostItv := {1..4};
set Products := {1..4};
set PriceItv := {1..3};
set Scale := {1..3};

param plant_capacity := 2000;

param ul := 0  30  60  90
            0 150 300 700
            0  40  80 150
            ;
param uu :=  0  60  90  150
            0 300 700 1000
            0  80 150  300
            ;
param vl := 0 200 500
            0 200 500
            0 100 300
            0 100 300
            ;
param vu := 200 500 1200
            200 500 1200
            100 300 1000
            100 300 600
            ;
param wl := 0 100 600
            0 100 600
```

```
            0 100 300
            0 100 300
            ;
param wu := 0 400 1000
            0 400 1000
            0 200 400
            0 200 400
            ;
param c := 200 160 120 100
           30   24  21  21
           120 100  90  80
           ;
param r := 18 15 12
           21 18 16
           28 24 20
           36 32 28
           ;
param a := 0.03 0.04 0.06 0.10
           0.15 0.18 0.20 0.28
           0.05 0.06 0.10 0.15
           ;
```

An MLLP Model for Configuration

B.1 THE MODEL

```
#
# File: Configuration
# Authors: Greger Ottosson, Erlendur S. Thorsteinsson
#
# Description: MLLP model of a configuration problem.
#              This problem has non-linear
#              quantity-cost term.

######### MODEL ##########

set Components;      # The components
set Types;           # Type of each component
set Resources;       # The resources/attributes

# Types of components
var T {t in Components} integer, := Types;
# Quantity of components
var Qt {c in Components} integer, shadowed, <= 10;
var R {r in Resources};

param a {r in Resources, c in Components, t in Types};
param limit {r in Resources};
param restr_comps;
param restr_types;
param num_types;

###### OBJECTIVE FUNCTION
minimize cost integral: R[1];
```

```
###### CONSTRAINTS

# The core of the problem. Note the variable
# subscript, i.e. variable 'T[c]' used as index
# in vector 'a', and that this is multiplied with
# the quantity variable 'Qt'.
subject to usage: forall {r in Resources}
        R[r] = sum {c in Components}
                   Qt[c]*a[r,c,T[c]];

subject to smaller_domain:
        forall {c in {1..restr_comps}}
           T[c] in {1..restr_types};

subject to resource: forall {r in Resources}
        R[r] >= limit[r];

subject to restriction1:
        T[1] in {1} -> T[2] in {1,2},
        T[3] in {1} -> (T[4] in {1,3} and
                        T[5] in {1,3,4,6} and
                        T[6] in {3} and
                        T[7] in {1,2});

var B boolean;
subject to quantity:
        (B in {1} -> (Qt[1] >= 1, Qt[2] = 0)),
        (B in {0} -> Qt[1] = 0);

######### SEARCH ##########

search {
        generate ordered by backfail
                 value by best triple_on_lp
                B
                {c in Components} T[c]
                {c in Components} Qt[c];
};
```

B.2 A DATA INSTANCE

```
######### CONFIGURATION DATA ##########
set Components := {1..8};
set Types := {1..10};
```

```
set Resources := {1..4};
param restr_comps := 4;
param restr_types := 5;
param num_types := 10;

param limit :=
0,100,0,0;

param a :=
1, 2, 3, 4, 5, 0, 0, 0, 0, 0,
1, 2, 3, 4, 5, 0, 0, 0, 0, 0,
1, 2, 3, 4, 5, 0, 0, 0, 0, 0,
1, 2, 3, 4, 5, 0, 0, 0, 0, 0,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

1, 3, 5, 7, 8, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
3, 4, 5, 6, 7, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 2, 3, 4, 5, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
4, 4, 4, 4, 5, 5, 5, 6, 6, 6,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
2, 2, 1, 1, 0, 0, 4, 4, 4, 4,

2, 2, 2, 2, 2, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
4, 3, 2, 2, 2, 0, 0, 0, 0, 0,
1, 1, 1, 1, 2, 2, 2, 2, 2, 2,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
```

Bibliography

The numbers inside braces indicate on which pages each citation occured.

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathemathical and Computer Modelling*, 17(7):57–73, 1993. {4, 14, 18, 52}

2. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993. {17}

3. D. Applegate and B. Cook. A Computational Study of the Job-shop Scheduling Problem. *Operations Research Society of America*, 3 (2), 1991. {17}

4. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *In Proceedings of Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, pages 73–87, 1999. {6}

5. E. Balas. Disjunctive programming: Cutting planes from logical conditions. *Nonlinear Programming 2*, pages 279–312, 1975. {44}

6. E. Balas. Disjunctive programming. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II, 5*, pages 3–51, Amsterdam, 1979. Annals of Discrete Mathematics, North-Holland. {44, 46, 93, 114}

7. P. Baptiste, C. Le Pape, and W Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *1st Joint Workshop on Artificial Intelligence and Operational Research*, 1995. {17}

8. C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998. {19}

9. Peter Barth and Alexander Bockmayr. Solving 0-1 problems in CLP($\mathcal{PB}$). In *Proceedings 9th Conf. Artificial Intelligence for Applications (CAIA)*, pages 263–269, Orlando, 1993. IEEE. {18}

10. Peter Barth and Alexander Bockmayr. Finite domain and cutting plane techniques in CLP($\mathcal{PB}$). In *International Conference of Logic Programming, ICLP'95*, Tokyo, 1995. {18}

11. Peter Barth and Alexander Bockmayr. Modelling mixed-integer optimisation problems in constraint logic programming. Technical Report MPI-I-95-2-011, Max-Planck-Institut fuer Informatik, Saarbruecken, 1995. {14}

12. M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. Wiley, New York, 1990. {39}

13. E.M.L. Beale and J.A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In John Lawrence, editor, *Proceeding of the Fifth International Conference on Operation Research*. Tavistock Publications, 1970. {17}

14. N. Beaumount. An algorithm for disjunctive programs. *European Journal of Operational Research*, 48:362–371, 1990. {44, 47}

15. Nicolas Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. Technical Report SICS T2000/01, SICS, 2000. {18}

16. Nicolas Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. {4, 18, 34, 52}

17. J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numer. Math.*, 4:238–252, 1962. {5, 76}

18. H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 8, pages 245–272. Elsevier, 1995. {15, 39, 75, 88}

19. U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, New York, 1972. {42}

20. A. Bockmayr and T. Kasper. Branch-and-infer: A unifying framework for integer and finite domain constraint programming. *INFORMS J. Computing*, 10(3):287 – 300, 1998. {16, 18, 26, 43, 75, 81, 88, 90}

21. A. Bockmayr and T. Kasper. A unifying framework for integer and finite domain constraint programming. Technical report, Max-Planc-Institut für Informatik, Saarbrücken, 1998. {62}

22. Alexander Bockmayr. Solving pseudo-boolean constraints. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995. (Châtillon-sur-Seine Spring School, France, May 1994). {18}

23. S. Bollapragada, O. Ghattas, and J. N. Hooker. Optimal design of truss structures by mixed logical and linear programming. *Operations Research, to appear*, 1995. {18, 26}

24. A. Brooke, D. Kendrick, and A. Meeraus. *GAMS-A User's Guide*. Scientific Press, San Francisco, 1992. {14, 52}

25. A. Brooke and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982. {52}

26. J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, pages 269–287, 1990. {17}

27. J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, pages 146–161, 1994. {17}

28. Mats Carlsson and Greger Ottosson. Anytime frequency allocation with soft constraints. In *CP96 Pre-Conference Workshop on Applications*, 1996. {18}

29. Mats Carlsson and Greger Ottosson. A comparison of CP, IP and hybrids for configuration problems. Technical report, Swedish Institute of Computer Science, 1999. {15, 16}

30. Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, sep 1997. {6}

31. Y. Caseau and F. Laburthe. Improved CLP Scheduling with Task Intervals. In *Proceedings of the Eleventh International Conference on Logic Programming*. MIT Press, 1994. {17}

32. Y. Caseau and F. Laburthe. Solving various weighted matching problems with constraints. In *Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 17–31, 1997. {5, 17, 34}

33. Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In Pascal Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, Massachusetts Institute of Technology, 1994. The MIT Press. {52}

34. Yves Caseau and François Laburthe. Solving small TSPs with constraints. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 316–330, Cambridge, July 8–11 1997. MIT Press. {5, 17, 34, 52, 57}

35. Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983. {115}

36. David P. Clements, James M. Crawford, David E. Joslin, George L. Nemhauser, Markus E. Puttlitz, and Martin W. P. Savelsbergh. Heuristic optimization: A hybrid AI/OR approach. In *Proceedings of the Workshop on Industrial Constraint-Directed Scheduling, in conjunction with the Third International Conference on Principles and Practice of Constraint Programming (CP97)*, 1997. {19}

37. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990. {97}

38. CPLEX. *CPLEX Manual*, 1998. URL http://www.cplex.com. {5, 100, 101}

39. G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963. {1}

40. Ken Darby-Dowman and James Little. The significance of constraint logic programming to operational research. *Operational Research Tutorial Papers*, pages 20–45, 1995. {14}

41. Ken Darby-Dowman and James Little. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal on Computing*, 10(3):276–286, Summer 1998. {14, 74}

42. Ken Darby-Dowman, James Little, Gautam Mitra, and Marco Zaffalon. Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem. *Constraints*, 1(3):245–264, March 1997. {13}

43. Dash Associates, Ltd. XPRESS-MP. Blisworth, Northants, England. {5, 14, 115}

44. M. Dawande and J. N. Hooker. Inference-based sensitivity analysis for mixed integer/linear programming. *Operations Research, to appear.* {36}

45. Bruno De Backer and Henri Beringer. Intellingent backtracking for CLP languages: An application to CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, editors, *ILPS'91: Proceedings International Logic Programming Symposium*, pages 405–419, San Diego, CA, October 1991. MIT Press. {84}

46. Bruno De Backer and Henry Beringer. A CLP language handling disjunctions of linear constraints. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 550–563, Budapest, Hungary, 1993. The MIT Press. {39}

47. Bruno De Backer, Vincent Furnon, Patrick Prosser, Philip Kilby, and Paul Shaw. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics special issue on Constraint Programming*, 1997. {19}

48. I. R. de Farias, E. L. Johnson, and G. L. Nemhauser. A branch-and-cut approach without binary variables to combinatorial optimization problems with continuous variables and combinatorial constraints. *Knowledge Engineering Review, special issue on AI/OR, submitted*, 1999. {17, 18, 26}

49. Mehmet Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *FGCS-88: Proceedings International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, December 1988. ICOT. {83}

50. H. El Sakkout, T. Richards, and M. Wallace. Minimal perturbance in dynamic scheduling. In Prade [115], pages 504–508. {18, 26, 99}

51. Mats Carlsson et al. SICStus Prolog User's Manual. SICS research report, Swedish Institute of Computer Science, 1995. URL: http://www.sics.se/sicstus. {83, 118}

52. Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*. Springer, October 1999. {5, 18, 110, 116, 119}

53. Filippo Focacci, Andrea Lodi, and Michela Milano. Integration of CP and OR methods for matching problems. In *CP-AI-OR'99 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, Feb 1999. {18, 110}

54. Filippo Focacci, Andrea Lodi, and Michela Milano. Solving TSP with time windows with constraints. In *Sixteenth International Conference on Logic Programming*, November 1999. {18, 26, 110, 116, 119}

55. Filippo Focacci, Andrea Lodi, Michela Milano, and Danielo Vigo. Solving TSP through the integration of OR and CP techniques. In *CP98 Workshop on Large Scale Combinatorial Optimisation and Constraints*, October 1998. {18, 110, 116, 119}

56. L.R. Foulds and J.M. Wilson. A variation of the generalized assignment problem arising in the New Zealand dairy industry. *Annals of Operations Research*, (69):105–114, 1997. {14}

57. R. Fourer. Extending a general-purpose algebraic modeling language to combinatorial optimization: A logic programming approach. In *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search*, Dordrecht, 1994. Kluwer. {14, 44, 52}

58. R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36:519–554, 1990. {52}

59. R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL – A Modeling Language for Mathematical Programming.* The Scientific Press, South San Francisco, 1993. {12, 14, 44, 52, 90, 97, 102}

60. Thom Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995. (Châtillon-sur-Seine Spring School, France, May 1994). {6}

61. R. Garfinkel and G. Nemhauser. Optimal political districting by implicit enumeration techniques. *Management Science*, 16, 1970. {34}

62. I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In E. C. Freuder, editor, *Principles and Practice of Constraint Programming*, pages 179–193. Springer, 1996. {3}

63. A.M. Geoffrion. Lagrangian relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974. {76}

64. Carmen Gervet. Large combinatorial optimization problems: A methodology for hybrid models and solutions. In *Journées Francophones De Programmation En Logique Et Par Contraintes*, 1998. {15}

65. Lise Getoor, Greger Ottosson, Markus Fromherz, and Björn Carlson. Effective redundant constraints for online scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI '97)*. American Association for Artificial Intelligence, July 1997. {5}

66. F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998. {2}

67. I. E. Grossmann, J. N. Hooker, R. Raman, and H. Yan. Logic cuts for processing networks with fixed charges. *Computers and Operations Research*, 21:265–279, 1994. {18, 26}

68. Vandecasteele H and Rodošek R. Modelling combinatorial problems for CLP(FD+R). In *Proceedings of the 10th Benelux Workshop on Logic Programming*, 1998. {18}

69. E. Hadjiconsantinou, C. Lucas, G. Mitra, and S. Moody. Tools for reformulating logical forms into zero-one mixed integer programs. *European Journal of Operational Research*, 1993. {17}

70. Mozafar Hajian, H. El-Sakkout, Mark Wallace, J.M.Lever, and Barry Richards. Towards a closer integration of finite domain propagation and simplex-based algorithms. Technical report, IC-Parc, 1995. {16}

71. Mozafar Hajian, Robert Rodošek, and Barry Richards. Introduction of a new class of variables to discrete and integer programming problems. *Baltzer Journals*, 1996. {17, 75}

72. Mozafar T. Hajian. Dis-equality constraints in linear/integer programming. Technical report, IC-Parc, 1996. {17, 75}

73. William Harvey and Matthew Ginsberg. Limited discrepancy search. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995. {3, 22}

74. Susanne Heipcke. Integrating constraint programming techniques into mathematical programming. In Prade [115], pages 259–260. {16, 18, 26, 88}

75. J. N. Hooker. Logic-based methods for optimization. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA). {26, 78, 87, 89}

76. J. N. Hooker. Inference duality as a basis for sensitivity analysis. In *Principles and Practice of Constraint Programming–CP96*, volume 1118 of *Lecture Notes in Computer Science*, 1996. {36}

77. J. N. Hooker. Inference duality as a basis for sensitivity analysis. *Constraints*, 4:101–112, 1999. {36}

78. J. N. Hooker. *Logic-Based Methods for Optimization.* Wiley, New York, 2000. {47}

79. J. N. Hooker and Hak-Jin Kim. Succinct relaxations for some discrete problems. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, 1998. {62}

80. J. N. Hooker and Hak-Jin Kim. Obtaining bounds from the relaxation dual. *In preparation*, 1999. {41}

81. J. N. Hooker and M. A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96–97(1–3):395–442, 1999. {14, 15, 18, 26, 44, 53, 54, 62, 78, 80, 87, 89}

82. John N. Hooker, Hak-Jin Kim, and Greger Ottosson. A declarative modeling framework that integrates solution methods. *Annals of Operations Research, Special Issue on Modeling Languages and Approaches, to appear*, 1998. {i, 26, 78, 87, 89}

83. John N. Hooker and Greger Ottosson. Logic-based benders decomposition. *In preparation.*, 1999. {38}

84. John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson, and Hak-Jin Kim. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 136–141. AAAI, The AAAI Press/The MIT Press, July 1999. {i, 26, 87, 88, 89, 110}

85. John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson, and Hak-Jin Kim. A scheme for unifying optimization and constraint programming. *Knowledge Engineering Review, special issue on AI/OR, to appear*, 1999. {i, 15, 87, 89, 92, 93, 110}

86. ILOG. *Using the CPLEX Callable Library*, 1997. www.cplex.com. {115}

87. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987. ACM. {2}

88. Vipul Jain and Ignacio E. Grossmann. Algorithms for hybrid MILP/CLP models for a class of optimization problems. Technical report, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, USA, September 1999. {16, 18, 22, 26}

89. Carsten Jordan and Andreas Drexl. A comparison of constraint and mixed-integer programming solvers for batch sequencing with sequence-dependent setups. *ORSA Journal on Computing*, 1995. {14}

90. Ulrich Junker, Stefan E. Karisch, Niklas Kohl, Bo Vaaben, Torsten Fahle, and Meinolf Sellmann. A framework for constraint programming based column generation. In *CP'99 Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1999. {19}

91. Philip Kay. *COSYTEC White Paper: CHIP Example Code*, April 1997. URL: www.cosytec.fr. {14}

92. Philip Kilby, Patrick Prosser, and Paul Shaw. A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraint. *Constraints special Issue on Industrial Constraint-directed Scheduling*, 1998. {19}

93. J.N. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Evolutionary training of CLP-constrained neural networks. In *2nd Int. Conf. on Practical Application of Constraint Technology (PACT'96)*, pages 129–142. The Practical Application Company Ltd, 1996. {19}

94. Ludwig Krippahl and Pedro Barahona. Applying constraint programming to protein structure determination. In *In Proceedings of Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, 1999. {6}

95. A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrika*, 28(3):497–520, 1960. {3}

96. Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998. {2, 5, 16, 29, 33, 44, 52, 83, 91, 97}

97. K. I. M. McKinnon and H.P. Williams. Constructing integer programming model by the predicate calculus. *Annals of Operations Research*, (21):227–246, 1989. {17}

98. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992. {19}

99. L. Michel and P. Van Hentenryck. Helios: A modeling language for global optimization and its implementation in Newton. *Theoretical Computer Science*, 173(1):3–48, February 1997. {63}

100. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992. {3}

101. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988. {1, 6, 19, 115, 117}

102. W. P. M. Nuijten and E. H. L. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*, 1995. Accepted for publication. {17}

103. Klas Orsvärn and Tomas Axling. The Tacton view of configuration tasks and engines. In *Workshop on Configuration, Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, 1999. URL http://www.tacton.com, Technical Report WS-99-05. {119}

104. Greger Ottosson and Mats Carlsson. Using global constraints for frequency allocation. Astec technical report, Uppsala University, Computing Science Dept., Aug 1996. {18}

105. Greger Ottosson, John N. Hooker, Hak-Jin Kim, and Erlendur S. Thorsteinsson. On integrating constraint propagation and linear programming for combinatorial optimization. In *CP-AI-OR'99 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, Feb 1999. {10, 110}

106. Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*. ACM, June 1997. {6}

107. Greger Ottosson and Erlendur S. Thorsteinsson. Linear relaxations and reduced-cost based propagation of continuous variable subscripts. In *CP-AI-OR'00 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, Mar 2000. {i}

108. Greger Ottosson, Erlendur S. Thorsteinsson, and John N. Hooker. Mixed global constraints and inference in hybrid CLP–IP solvers. In Susanne Heipcke and Mark Wallace, editors, *CP99 Post-Conference Workshop on Large Scale Combinatorial Optimisation and Constraints*, volume 4 of *Electronic Notes in Discrete Mathematics*. Elsevier Science, October 1999. {i, 18, 26, 27, 110}

109. Franccois Pachet and Pierre Roy. Automatic generation of music programs. In *In Proceedings of Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, 1999. {6}

110. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization (Algorithms and Complexity)*. Prentice-Hall, New Jersey, 1982. {1}

111. P. M. Pardalos and J. B. Rosen. *Constrained global optimization: Algorithms and Applications*. Lecture Notes in Computer Science 268. Springer-Verlag, 1987. {63}

112. G. Pesant and M. Gendreau. A view of local search in constraint programming. *Lecture Notes in Computer Science*, 1118:353–366, 1996. {19}

113. G. Pesant and M. Gendreau. A constraint programming framework for local search methods. 1999. {19}

114. J. M. Pinto and I. E. Grossmann. A logic-based approach to scheduling problems with resource constraints. *Computers and Chemical Engineering*, 21:801–818, 1997. {18, 26}

115. Henri Prade, editor. *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*. John Wiley & Sons, 1998. {134, 136}

116. L. Proll and B. Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS Journal on Computing*, 10:265–275, 1998. {14}

117. Jean-Franccois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *ILPS'95: Proceedings 5rd International Logic Programming Symposium*, pages 513–527. MIT Press, 1995. {17}

118. R. Raman and I. E. Grossmann. Relation between MILP modeling and logical inference for chemical process synthesis. *Computers and Chemical Engineering*, 15:73–84, 1991. {18, 26}

119. R. Raman and I. E. Grossmann. Modeling and computational techniques for logic based integer programming. *Computers and Chemical Engineering*, 18:563–578, 1994. {18, 26}

120. Colin R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Halsted Press, New York, 1993. {2}

121. Philippe Refalo. Tight cooperation and its application in piecewise linear optimization. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*. Springer, October 1999. {16, 18, 26}

122. J-C Régin. A filtering algorithm for constraints of difference in CSPs. Technical Report R.R.LIRMM 93-068, LIRM, Dec 1993. {34}

123. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994. {18, 29, 52}

124. J.-C. Régin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 32–41, 1997. {34}

125. Jean-Charles Régin. Arc consistency for global cardinality constraints with costs. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*. Springer, October 1999. {34}

126. Robert Rodošek and Mark Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In *Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 385–399, 1998. {18, 26}

127. Robert Rodošek, Mark Wallace, and Mozafar Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Baltzer Journals*, 1997. {15, 18, 26, 62, 75, 88, 99, 109}

128. A. Ruiz-Andino and J. J. Ruz. Labeling in CLP(FD) with evolutionary programming. In *Proceedings of the Joint Conference on Declarative Programming, GULP-PRODE'95*, pages 569–580, 1995. {19}

129. A. Ruiz-Andino and J. J. Ruz. Integration of CLP and stochastic optimisation strategies. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 351–352. MIT Press, 1998. {19}

130. M. W. P. Savelsbergh. Preprocessing and probing for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994. {5, 6, 16}

131. P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–??, 1998. {18}

132. Barbara Smith, Sally Brailsford, Peter Hubbard, and H. Paul Williams. The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. In *CP95: Proceedings 1st International Conference on Principles and Practice of Constraint Programming)*, Marseilles, September 1995. {13, 14, 74}

133. P. J. Stuckey and V. Tam. Models for using stochastic constraint solvers in constraint logic programming. *Lecture Notes in Computer Science*, 1140:423–??, 1996. {19}

134. Tacton Systems. *Tacton Configurator User's Manual*, 1999. URL http://www.tacton.com. {119}

135. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993. {3, 5, 37, 42, 84}

136. M. Türkay and I. E. Grossmann. Logic-based MINLP algorithms for the optimal synthesis of process networks. *Computers and Chemical Engineering*, 20:959–978, 1996. {18, 26}

137. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989. {2, 44, 52}

138. Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999. {15, 18, 26, 44, 52, 97, 99, 101, 103}

139. Pascal Van Hentenryck and Laurent Michel. Newton: Constraint programming over nonlinear real constraints. Technical Report CS-95-25, Department of Computer Science, Brown University, August 1995. Sun, 13 Jul 1997 18:30:15 GMT. {63}

140. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995. (Châtillon-sur-Seine Spring School, France, May 1994). {2, 52}

141. F. Vanderbeck. Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming*, 86(3):565–594, 1999. {19}

142. Joachim P. Walser. *Integer optimization by local search: a domain-independent approach*, volume 1637 of *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*. Springer-Verlag Inc., New York, NY, USA, 1999. {14, 19}

143. H. P. Williams. Logic problems and integer programming. *Bulletin of the Institute of Mathematics and its Applications*, (13):18–20, 1977. {17}

144. H. P. Williams. *Model Building in Mathematical Programming.* Wiley, New York, second edition, 1985. {6}

145. H. P. Williams. Linear and integer programming applied to the propositional calculus. *International Journal of Systems Research and Information Science*, (2):81–100, 1987. {17}

146. L. A. Wolsey. *Integer Programming.* John Wiley, New York, 1998. {1, 5, 19, 22, 116}

147. Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In G. Görz and S. Hölldobler, editors, *20th German Annual Conference on Artificial Intelligence*, volume 1137 of *LNAI*, pages 377–386, Dresden, Germany, 1996. Springer-Verlag. {6}

Computing Science Department
Information Technology
Uppsala University
Box 311, SE–751 05 Uppsala, Sweden

Uppsala Theses in Computer Science 33
ISSN 0283–359X
ISBN 91–506–1396–0