On the Integration of Optimization Techniques

Tallys H. Yunes

A dissertation submitted in partial fulfilment of the
requirements for the degree of Doctor of Philosophy at the
Tepper School of Business, Carnegie Mellon University.
Pittsburgh, PA 15213. U.S.A.

On The Integration of Optimization Techniques

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the Tepper School of Business, Carnegie Mellon University.

Thesis committee:

John Hooker (Chair), Tepper School of Business, Carnegie Mellon University
Ignacio Grossmann, Chemical Engineering Department, Carnegie Mellon University
Sridhar Tayur, Tepper School of Business, Carnegie Mellon University
Michael Trick, Tepper School of Business, Carnegie Mellon University
Pascal Van Hentenryck, Computer Science Department, Brown University

This research was supported by a William Larimer Mellon Fellowship,
the National Science Foundation under grant ACI - 0121497, and John Deere & Co.

Abstract

The recent literature has numerous examples showing that the combination of different optimization techniques can outperform traditional approaches on a variety of optimization problems. This thesis is composed of four papers in which the combination of Mixed Integer Linear Programming (MILP) and Constraint Programming (CP) is exploited to a greater or lesser extent.

In the first paper, we identify a practical application for the global constraint *sum*, called the Sequence Dependent Cumulative Cost Problem (SDCCP), and we describe the convex hull of feasible solutions to *sum*. We also show that, from both a theoretical and computational point of view, models that use the *sum* constraint are superior to an alternative model for the SDCCP.

In the second and third papers we design, implement and test an integrated modeling and solution system, which we call SIMPL. SIMPL's main idea is to view traditional solution methods, as well as integrated methods, as special cases of a *search-infer-and-relax* framework. Computational experiments on three different problem classes indicate that SIMPL is generic enough to allow its user to implement a variety of complex integrated solution schemes with a high-level and concise modeling language. This means that, in many cases, it is no longer necessary to write special purpose code in order to exploit many of the benefits of an integrated approach. In two examples, our experiments show reductions in the number of search nodes and computation time, respectively, of a few orders of magnitude.

The fourth paper deals with a problem at the intersection of marketing and operations. We have proposed and implemented a new approach combining customer behavior and product line optimization that has helped John Deere & Co. reduce the complexity cost of its tractor lines. Unlike previous approaches reported in the literature, our approach is capable of handling very large practical problems. As a result of this effort, John Deere & Co. expects to save tens of millions of dollars annually.

To Renata, with love.

Acknowledgments

This dissertation is not only the result of over five years of work, but also the result of a remarkable life experience I had while at Carnegie Mellon University. I want to express my sincere gratitude to all of those who played a role in this journey.

I would like to deeply thank my advisor John Hooker for giving me the wonderful opportunity to work with and learn so much from him. John Hooker's unique way of understanding (and explaining) mathematics and operations research will be an everlasting source of inspiration to my future research career. Thank you, John, for the invaluable pieces of advice, and for always being considerate, patient, and so unbelievably generous with your time.

I am also indebted to professors Ignacio Grossmann, Sridhar Tayur, Michael Trick and Pascal Van Hentenryck for their guidance and expert advice on improving the quality of this dissertation.

I owe a great deal of my intellectual capital to many faculty members at the Tepper School. I am much obliged to Egon Balas, Gérard Cornuéjols, Javier Peña, R. Ravi, Alan Scheller-Wolf, and Michael Trick for the the enriching lectures, enlightening technical discussions, and personal conversations.

My journey as a PhD student would not have been the same without the great friends I met at Carnegie Mellon. Thanks to Nihat Altintas, Kent Andersen, Ionuţ Aron, Atul Bhandari, Borga Deniz, Michele Di Pietro, Espen Henriksen, Miroslav Karamanov, Jochen Könemann, Yanjun Li, Manojkumar Puthenveedu, Anureet Saxena, Amitabh Sinha, and Erlendur Thorsteinsson for all the enjoyable moments, constant support, and sincere friendship. Special thanks go to my co-author, Ionuţ, for the numerous hours of productive, insightful and *fun* research. It was just the beginning, my friend.

I cannot forget to say a few words about those who steered my fate toward landing in Pittsburgh in early August 2000. Thanks to Cid de Souza for not letting me get a job in industry and for introducing me to Integer Programming. Thanks to Arnaldo Moura for introducing me to Constraint Programming. And, of course, thanks to Guido Araújo for stopping me in the middle of the street in 1999 and asking "Have you already applied to Carnegie Mellon?"

Sincere thanks go to my mother, Osmarina, and my father, Tadeu, for their love and encouragement. We are far apart, but you are always in my heart.

Last, but definitely not least, I would like to thank my lovely wife Renata. Your constant support and unconditional love cannot be measured. Your joyful spirit brightens and lightens my days. You have taught me to be a better man, never letting me forget about what really matters. This thesis is dedicated to you my *querida, amore, princesa, fofucha, nenê, bochecha, mamosinho, pesseguinho.*

Contents

List of Figures

List of Tables

Chapter 1

Summary

1.1 Introduction

There is rarely a day in our lives when we are not faced with *decisions*. Such decisions can be simple, such as which clothes to wear, or they can be more complicated such as which of two job offers to accept. Managers of all levels and in different sectors of our economy often need to make decisions that can have a serious impact on their companies.

We classify decision problems in two categories: *feasibility* and *optimization*. Feasibility problems are concerned with questions of the form *"Is it possible to achieve this goal?"* (e.g. Can we place 8 queens on a chessboard so that no two queens can attack each other?), while optimization problems try to answer questions of the form *"What is the best way to achieve this goal?"* (e.g. In which order should the mailman deliver today's letters in order to walk the smallest possible distance?).

Operations Research (OR) is a branch of science that is concerned with using analytical methods (i.e. rigorous mathematical techniques) to solve decision problems. The decisions that have to be made are called *variables*. Our decisions are usually limited or restricted by some rules (e.g. limited amount of time and/or money, physical restrictions, laws, political interests, etc.). Such restrictions on the values that can be assigned to the decision variables are called *constraints*. Finally, the goal of an optimization problem is called an *objective function*.

Traditionally, when a decision problem is complicated (i.e. cannot be solved with pencil and paper), specialists use computers to solve them. Because computers cannot think by themselves (yet), they need two things: a precise description of the decision problem (a.k.a. a *model*) and a recipe to solve the problem (a.k.a. an *algorithm*). Usually, there are many different models to represent the same decision problem. Moreover, given a model, there can be many different algorithms to solve it. The question of choosing a *good* pair of model and algorithm is a very difficult and fundamental question.

1.2 Some Traditional Modeling and Solving Paradigms

Linear Programming (LP) (Dantzig, 1963) is a very popular approach to solving certain types of optimization problems. In LP, variables take real (i.e. continuous) values. The objective function and constraints on the variables have to be linear expressions. In many circumstances, however, it is important to require that some variables in an optimization model assume integral values. For example, variable x may represent the number of pilots an airline company has to hire, or y may represent whether or not a bridge should be built (in this case y is said to be a *binary* variable because it can only take two values: 0 means "do not build the bridge"; 1 means "build it"). When an LP model has the additional integrality requirement on some (or all) of its variables, it is called a Mixed

Integer Linear Programming (MILP) model (Nemhauser and Wolsey, 1988; Wolsey, 1998).

During the 1980's and 1990's another approach to modeling/solving feasibility and optimization problems, called Constraint Programming (CP) (Van Hentenryck, 1989; Marriott and Stuckey, 1998), gained a lot of popularity. CP originated in the Computer Science (CS) and Artificial Intelligence (AI) communities, and the way it represents and solves feasibility and optimization models is very different from LP and MILP. While algorithms for LP and MILP models are more strongly guided by the optimization aspect of the problem, CP algorithms are guided by feasibility. The main idea in CP is to try to reduce the sets of possible values for the problem variables (called *variable domains*) by performing logical inferences. These logical inferences or deductions are called *domain reductions* and they are based on the way variables are related to each other through constraints. For instance, if $x \in \{1, \ldots, 10\}$, $y \in \{2, \ldots, 8\}$ and we need $2x + y >= 25$, we can then conclude that $x \geq 9$ and $y \geq 5$. Deductions are performed locally, constraint by constraint. Once all constraints have been examined, the process is repeated because the latest deductions (i.e. reduced variable domains) may enable some constraints to deduce other facts. This whole mechanism is called *constraint propagation*. An additional interesting aspect about CP, relative to MILP, is the fact that its constraints are not restricted to linear functions.

Another important approach to solving optimization problems stems from the observation that a good-enough solution (as opposed to *the best* or *optimal* solution) may be sufficient for our purposes, if finding the best solution is impossible or too time consuming. In this case, the solution algorithm is called a *heuristic* (see Reeves 1993 for examples).

The recent literature (see Section 1.4) has numerous examples showing that some types of optimization problems are better solved when ideas from MILP, CP and heuristics are combined, rather than used separately. In broad terms, this happens because, although each of these alternative solution techniques has strengths and weaknesses, they complement each other. Therefore, when a problem has features that can benefit from the strengths of two or more techniques, their combination may yield very good results.

In this thesis we investigate the topic of combining alternative optimization techniques into integrated approaches for solving optimization problems. Our contributions to this field address different questions (see Section 1.5), including the issue of how to make integrated approaches easier to use.

1.3 The Ubiquity of Search, Inference and Relaxation

When we analyze the way optimization problems are solved by traditional techniques, it is possible to identify many commonalities. More specifically, the following three steps are usually present:

Search. Looking for a feasible or optimal solution involves exploring the collection of feasible assignments of values to the variables. This is referred to as the *feasible region* or *search space* of the problem. Given the search space is often huge, solution algorithms tend to break it in pieces (*restrictions*) and look at some (or all) of the pieces according to a specified order. In addition, solution algorithms also try to assign values to the decision variables.

Inference. As the search step proceeds, problem restrictions are created and variables receive tentative values. As a result of this, it may be possible to deduce new facts about the search space. For instance, if we are in a region of the search space where variable x has to be zero, and we know that x and y are related to each other through some constraints, we may be able to conclude that $y \geq 5$. Inferences of this sort can help accelerate the exploration of the search space.

Relaxation. The only way to guarantee that an optimal solution to a decision problem has been found is to show that its value is equal to a given *bound* on the optimal value. For example, if physicians show that it is physically impossible for a human being to run 100m in less than 8 seconds (a *lower bound*) and we watch an athlete do it in 8.3 seconds, we can be sure that this person is doing very well.

To obtain bounds for optimization problems, we solve *relaxations* of those problems. A relaxation is just a simpler and easier-to-solve version of the problem whose feasible region contains the original problem's feasible region. Therefore, the optimal value of the relaxation, which can be "easily" calculated, is a bound on the optimal value of the original problem.

To make this point more concrete, Table 1.1 (adapted from Hooker, 2005b) includes a list of some well known solution methods to optimization problems and how they define restrictions (for search), inference and relaxation. CGO stands for Continuous Global Optimization (Sahinidis and Tawarmalani, 2003); Benders stands for the Benders decomposition approach (Benders, 1962) and the logic-based Benders decomposition approach (Hooker and Ottosson, 2003); DPL stands for the Davis-Putnam-Loveland algorithm for satisfiability problems (Davis, Logemann, and Loveland, 1962); and Tabu Search is a widely used type of heuristic approach (Glover, 1986).

Table 1.1: The ubiquity of search, inference and relaxation.

Solution Method	Restriction	Inference	Relaxation
MILP	Branch on fractional vars.	Cutting planes, preprocessing	LP relaxation
CP	Split variable domains	Domain reduction, propagation	Current domains
CGO	Split intervals	Interv. propag., lagr. mult.	LP or NLP relaxation
Benders	Subproblem	Benders cuts (nogoods)	Master problem
DPL	Branching	Resolution and conflict clauses	Processed clauses
Tabu Search	Current neighborhood	Tabu list	Same as restriction

We can, therefore, view solution methods as a special case of a more general method that iterates over the above three generic steps. We call this general method a *search-infer-and-relax* framework (Hooker, 2005c). As a consequence, hybrid (or integrated) methods for optimization problems are simply *another* special case of that framework. In addition, it is important to note that these three steps can interact and provide useful information to each other (see Hooker, 2005c).

1.4 Previous Work

Comprehensive surveys of hybrid methods that combine CP and MILP are provided by Hooker (2000, 2002), and tutorial articles may be found Milano (2003). Various elements of the search-infer-and-relax framework presented here were proposed by Hooker (1994, 1997, 2000, 2003), Bockmayr and Kasper (1998), Hooker and Osorio (1999), and Hooker et al. (2000). An extension to dynamic backtracking and heuristic methods is given in Hooker (2005c).

Some of the concepts that are most relevant to the work presented in this thesis are: decomposition approaches that solve parts of the problem with different techniques (Eremin and Wallace, 2001; Hooker, 2000; Hooker and Ottosson, 2003; Hooker and Yan, 1995; Jain and Grossmann, 2001; Thorsteinsson, 2001); allowing different models/solvers to exchange information (Rodošek et al., 1999);

using linear programming to reduce the domains of variables or to fix them to certain values (Beringer and de Backer, 1995; Focacci et al., 1999; Rodošek et al., 1999); automatic reformulation of global constraints as systems of linear inequalities (Refalo, 2000); continuous relaxations of global constraints and disjunctions of linear systems (Balas, 1998; Hooker, 2000; Hooker and Osorio, 1999; Hooker and Yan, 2002; Ottosson et al., 1999; Williams and Yan, 2001; Yan and Hooker, 1999; Yunes, 2002); understanding the generation of cutting planes as a form of logical inference (Bockmayr and Eisenbrand, 2000; Bockmayr and Kasper, 1998); strengthening the problem formulation by embedding the generation of valid cutting planes into CP constraints (Focacci et al., 2000); maintaining the continuous relaxation of a constraint updated when the domains of its variables change (Refalo, 1999); and using global constraints as a key component in the intersection of CP and OR (Milano et al., 2002).

Existing hybrid solvers include ECL^{i}PS^e, OPL Studio, Mosel, and SCIP. ECL^{i}PS^e is a Prolog-based constraint logic programming system that provides an interface with linear and MILP solvers (Rodošek, Wallace, and Hajian 1999; Cheadle et al. 2003; Ajili and Wallace 2003). The CP solver in ECL^{i}PS^e communicates tightened bounds to the MILP solver, while the MILP solver detects infeasibility and provides a bound on the objective function that is used by the CP solver. The optimal solution of the linear constraints in the problem can be used as a search heuristic. OPL Studio provides an integrated modeling language that expresses both MILP and CP constraints (Van Hentenryck et al., 1999). It sends the problem to a CP or MILP solver depending on the nature of constraints. A script language allows one to write algorithms that call the CP and MILP solvers repeatedly. Mosel is both a modeling and programming language that interfaces with various solvers, including MILP and CP solvers (Colombani and Heipcke, 2002, 2004). SCIP is a callable library that gives the user control of a solution process that can involve both CP and MILP solvers (Achterberg, 2004).

1.5 This Work

We can summarize the main contributions of this thesis as follows.

- We have identified a practical application for the global constraint *sum*, called the Sequence Dependent Cumulative Cost Problem (SDCCP), and we have described the convex hull of feasible solutions to *sum* (Paper I). We have also shown that, from both a theoretical and computational point of view, models that use the *sum* constraint are superior to an alternative model for the SDCCP. Paper I contributes to the overall project of integrating methods by providing an OR-style relaxation of a CP-style constraint, and it advances the state of the art in CP.

- We have designed and implemented a system, called SIMPL, which is both a modeling language and a solver for integrated optimization problems (Papers II and III). SIMPL is an implementation of the *search-infer-and-relax* framework, which can be seen as a mature version of an extensive body of work that spans more than a decade of research (Hooker, 1994, 1997, 2000, 2003; Hooker and Osorio, 1999; Hooker, Ottosson, Thorsteinsson, and Kim, 2000; Bockmayr and Kasper, 1998);

- We have shown that SIMPL is generic enough to allow its user to implement a variety of complex integrated solution schemes (see Table 4.1) with a high-level and concise modeling language. This means that, in many cases, it is no longer necessary to write special purpose code in an imperative language in order to exploit many of the benefits of an integrated approach (Papers II and III);

- We have developed the idea of *constraint based control*, according to which constraints play the central role in an integrated model because they possess all the information necessary to perform inferences, create problem relaxations and guide the search (Papers II and III);

- We have used SIMPL to solve models from three distinct problem domains, showing evidence that the search-infer-and-relax framework for integrated problem solving advances the current state-of-the-art of high level modeling systems (Paper III);
- We have proposed and implemented a new approach combining customer behavior and product line optimization that has helped John Deere & Co. reduce the complexity cost of its tractor lines (Paper IV). Unlike previous approaches reported in the literature, our approach is capable of handling very large practical problems. As a result of this effort, John Deere & Co. expects to save tens of millions of dollars annually.

The following sections give a brief summary of the contents of each of the four papers included in this dissertation.

1.5.1 Paper I: On the Sum Constraint: Relaxation and Applications

The global constraint *sum* can be used as a tool to implement summations over sets of variables whose indices are not known in advance, as follows. Let $S_{j_1}, \ldots, S_{j_d}$ be sets of indices in $\{1, \ldots, n\}$, and let $c_1, \ldots, c_n$ be constants. If y is a variable with domain $D_y = \{j_1, \ldots, j_d\}$, the constraint $\text{sum}(y, (S_{j_1}, \ldots, S_{j_d}), (c_1, \ldots, c_n), z)$ states that

$$z = \sum_{j \in S_y} c_j \ .$$

An extended version of this constraint exists where each constant c_i is replaced by a variable x_i. One can think of the well known element constraint as a special case of the sum constraint in which we have $S_j = \{j\}$ for all $j \in \{1, \ldots, n\}$.

When studying a scheduling problem in the context of the development of new products in the agrochemical and pharmaceutical industries, we were faced with a subproblem where the sum constraint appears as a natural modeling candidate. We baptize this problem as the *Sequence Dependent Cumulative Cost Problem* (SDCCP). The difficulties faced by previous attempts to solve the original main problem (Honkomp et al., 1997; Jain and Grossmann, 1999) indicate that a combination of Mathematical Programming and Constraint Programming methodologies may be successful.

This paper has three main contributions. Firstly, from a hybrid modeling perspective, we give a first step toward better understanding the set of feasible solutions of the sum constraint by presenting its convex hull relaxation. Secondly, we look at two alternative formulations of the SDCCP, one using the sum constraint and another using the element constraint, and we show that the former is better than the latter in a well defined sense, i.e. its linear relaxation gives a tighter bound. Thirdly, our computational results for the SDCCP indicate that a CP model that uses the sum constraint also has the advantage of being more natural, more concise and computationally more efficient than the alternative model with the element constraint.

This paper appeared in the proceedings of the Eigth International Conference on the Principles and Practice of Constraint Programming (CP) and was published in *Lecture Notes in Computer Science*, volume 2470, pages 80–92, 2002 (Yunes, 2002).

1.5.2 Paper II: SIMPL: A System for Integrating Optimization Techniques

In recent years, the Constraint Programming (CP) and Operations Research (OR) communities have explored the advantages of combining CP and OR techniques to formulate and solve combinatorial optimization problems. These advantages include a more versatile modeling framework and the ability to

combine complementary strengths of the two solution technologies. Examples of existing programming languages that provide mechanisms for combining CP and OR techniques are ECL^{i}PS^e (Rodošek et al., 1999; Wallace et al., 1997), OPL (Van Hentenryck et al., 1999) and Mosel (Colombani and Heipcke, 2004).

Hybrid methods tend to be most effective when CP and OR techniques interact closely at the micro level throughout the search process. To achieve this one must often write special-purpose code, which slows research and discourages broader application of integrated methods. We address this situation by introducing a system for integrated modeling and solution called SIMPL (Modeling Language for Integrated Problem Solving). The SIMPL modeling language formulates problems in such a way as to reveal problem structure to the solver. The solver executes a search algorithm that invokes CP and OR techniques as needed, based on problem characteristics.

The design of such a system presents a significant research problem in itself, since it must be flexible enough to accommodate a wide range of integration methods and yet structured enough to allow high-level implementation of specific applications. Our approach, which is based partly on a proposal in Hooker (2000, 2003), is to view CP and OR techniques as special cases of a single method rather than as separate methods to be combined. This overarching method consists of an infer-relax-restrict cycle in which CP and OR techniques may interact at any stage.

In this paper we: briefly review some of the fundamental ideas related to the combination of CP and OR that are relevant to the development of SIMPL; describe the main concepts behind SIMPL and talk about implementation details; present a few examples of how to model optimization problems in SIMPL, explaining the syntax and semantics of the language; and outline some additional features provided by SIMPL.

This paper was co-authored with Ionuţ Aron and John Hooker. It appeared in the proceedings of the Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR) and was published in *Lecture Notes in Computer Science*, volume 3011, pages 21–36, 2004 (Aron, Hooker, and Yunes, 2004).

1.5.3 Paper III: An Integrated Solver for Optimization Problems

Note: Paper II concentrated on describing SIMPL's architecture, internal implementation and modeling language in some detail. The development status at the time of writing was such that only very small models had been solved, and the system was still under a lot of testing. Paper III builds on the ideas of Paper II, develops them to a more mature level, and presents more substantial computational experiments.

One of the central trends in the optimization community over the past several years has been the steady improvement of general-purpose solvers. Such mixed integer solvers as CPLEX and XPRESS-MP have become significantly more effective and robust, and similar advancements have occurred in continuous global optimization (BARON, LGO) and constraint programming (CHIP, ILOG Solver). These developments promote the use of optimization and constraint solving technology, since they spare practitioners the inconvenience of acquiring and learning different software for every application.

A logical next step in this evolution is to combine mixed integer linear programming (MILP), global optimization, and constraint programming (CP) in a single system. This not only brings together a wide range of methods under one roof, but it allows users to reap the advantages of integrated problem solving. Our ultimate goal in implementing SIMPL is to build an integrated solver that can be used as conveniently as current mixed integer, global and constraint solvers. We have attempted to address this situation by designing an architecture that achieves low-level integration of solution techniques

with a high-level modeling language.

SIMPL is based on two principles: algorithmic unification and constraint-based control. *Algorithmic unification* begins with the premise that integration should occur at a fundamental and conceptual level, rather than postponed to the software design stage. Optimization methods and their hybrids should be viewed, to the extent possible, as special cases of a single solution method that can be adjusted to exploit the structure of a given problem. We find that a *search-infer-and-relax* framework serves this purpose. It encompasses a wide variety of methods, including branch-and-cut methods for integer programming, branch-and-infer methods for constraint programming, popular methods for continuous global optimization, such nogood-based methods as Benders decomposition and dynamic backtracking, and even heuristic methods such as local search and greedy randomized adaptive search procedures (GRASPs).

Constraint-based control allows the design of the model itself to tell the solver how to combine techniques so as to exploit problem structure. Highly-structured subsets of constraints are written as metaconstraints, which are similar to "global constraints" in constraint programming. Each metaconstraint invokes relaxations and inference techniques that have been devised for its particular structure. Constraints also control the search. If a branching method is used, for example, then the search branches on violated metaconstraints, and a specific branching scheme is associated with each type of constraint.

The selection of metaconstraints to formulate the problem determines how the solver combines algorithmic ideas to solve the problem. This means that SIMPL deliberately sacrifices independence of model and method: the model must be formulated with the solution method in mind. However, we believe that successful combinatorial optimization leaves no alternative. This is evident in both integer programming and constraint programming.

We focus here on branch-and-cut, branch-and-infer, and generalized Benders methods, since these have been implemented so far in SIMPL. The system architecture is designed, however, for extension to continuous global optimization, general nogood-based methods, and heuristic methods.

We present three examples that illustrate some modeling and algorithmic ideas that are characteristic of integrated methods. A production planning problem with semicontinuous piecewise linear costs illustrates metaconstraints and the interaction of inference and relaxation. A product configuration problem illustrates variable indices and how further inference can be derived from the solution of a relaxation. Finally, a planning and scheduling problem shows how a Benders method fits into the framework. We show how to model the three example problems in SIMPL and include computational results.

This paper was co-authored with Ionuţ Aron and John Hooker, and it was submitted for publication in December 2005.

1.5.4 Paper IV: Building Efficient Product Portfolios at John Deere

Deer & Company (Deere) manufactures equipment for construction, commercial, and consumer applications, as well as engines and power train components. As a major player in many equipment markets, Deere maintains multiple product lines. Within each line, there may be several thousand, to several million, different product variants. Variants are built by selecting, for each *feature* available on a machine – e.g. engine type, transmission, axle – one of a number of possible *options* – e.g. 200, 250 or 300 horsepower (HP) for engines. Not all options are compatible, and a feasible combination of options is called a *configuration*.

Deere speculates that maintaining too many configurations reduces profits, by elevating what Deere calls *complexity cost*. This cost, over and above the inventory carrying costs of each configuration,

captures factors such as reduced manufacturing efficiency, frequent line changeovers, and the general overhead of maintaining documentation and support for a configuration.

In this paper we describe the marketing and operational methodology and tools we developed to reduce Deere's complexity costs, by concentrating product line configurations while maintaining high customer service, thus elevating overall profits. We illustrate our work with applications to two tractor lines at Deere. Details of the products have been disguised, but the lines differ in significant ways (costs, profits, sales), making them a diverse test bed for our optimization algorithm.

A primary component in our algorithm is our *customer migration model*, quantifying the behavior of Deere's customers. A customer may want a specific configuration, but if his/her first choice is unavailable he/she may *migrate* to an alternative configuration that does not differ too greatly from the first choice. Using actual sales, along with customer segmentations and part worths utilities provided by Deere, we probabilistically model every customer as *individually* identifying a set of acceptable configurations, sorted in decreasing order of preference. This sorted list of choices is the customer's *migration list*. When the top configuration on his/her list is not available, a customer will buy the next available configuration. When no configuration is available, the customer *defects* to a competitor.

Using each customer's migration list, as well as costs and profits for all feasible configurations, we build a Mixed Integer Program to maximize Deere's profits within a product line. Application of our algorithm provides: (i) A general method to determine which configurations are least profitable, and thus may be candidates for elimination; (ii) Recommendations for how Deere can significantly focus specific product lines; and (iii) Identification of the high level drivers of product line efficiency. Based on our results, Deere has instituted an incentive program to steer customers towards a core subset of their product lines, which has resulted in increased profit in line with our predictions – tens of millions of dollars annually (see Section 5.7 for details).

More specifically, in this paper we: describe how we generate and cost out all of the feasible configurations on a product line; detail how we generate customer utilities and migration lists; present results of the experiments on the Deere's product lines; and briefly describe Deere's actual implementation.

This paper was co-authored with Dominic Napolitano, Alan Scheller-Wolf and Sridhar Tayur. It was submitted for publication in May 2004 and re-submitted, after one review, in September 2005.

1.6 Future Work

I have closely collaborated with Ionuţ Aron in the design and implementation of SIMPL. We intend to continue working on SIMPL with the ultimate goal of making it available to a larger body of users besides our research group. The following are a few more specific research directions that stem from the work done in this dissertation.

- In Paper I we propose a convex hull relaxation for the sum constraint but we have not had the opportunity to test an integrated model that uses it. For example, it would be interesting to model the new product development problem with the sum constraint and run experiments to test whether it is helpful or not.

  In addition, it seems that a few commercial CP solvers have some sort of sum constraint in their library of global constraints (e.g. *IloPack* in ILOG Solver, *weight* in ECL^{i}PS^e). This may be an indication that this constraint has been used in other contexts of which we may not be aware. After my presentation of Paper I in the CP 2002 conference, I was approached by a person who said their company is dealing with a certain type of bin packing problem in which sum could be used. Unfortunately, he was not authorized to give me further details about the problem.

- The further development of SIMPL (Papers II and III) points to many interesting research topics:
  - Incorporate local search capabilities in SIMPL and modify/expand the modeling language accordingly;
  - Expand SIMPL's functionalities to allow for the high-level implementation of other types of successful integrated approaches, such as CP-based branch-and-price;
  - Incorporate a non-linear solver in SIMPL;
  - Increase SIMPL's library of global constraints. This includes developing (and implementing) new linear relaxations, inference modules and branching modules for those constraints;
  - Expand the expressiveness of the `SEARCH` section of a SIMPL model to allow for a finer control of the search, similar to what can be done in OPL.
  - Incorporate general purpose cutting planes in SIMPL, such as the ones currently used by CPLEX and XPRESS (e.g. Gomory cuts);
  - Enhance SIMPL's compiler to try to automatically identify useful (or harmful) structures in the model, which were possibly overlooked by the user, and exploit them if possible (e.g. symmetry). This topic was proposed by Pascal Van Hentenryck at the end of my thesis proposal;
- In Paper III we emphasize that SIMPL is still a research code and has not been optimized for speed. Nevertheless, this sort of code optimization is necessary both to increase our own productivity and to make SIMPL more appealing to other users. Some speed up can be obtained by simply using standard code-profiling tools such as Valgrind (Nethercote and Seward, 2003). We also intend to re-analyze and, if necessary, re-implement some of SIMPL's internal data structures.
- Paper IV can be extended in at least three ways:
  - It would be interesting to use our methodology to study whether complexity costs (or variants thereof) can be reduced in other sectors of industry where product variety abounds, e.g. food and electronics.
  - One advantage of our customer migration model is that it does not care about how the customer migration lists are obtained. We propose one way of generating the lists, which may or may not be adequate for a given application area. From a marketing and/or cognitive psychology perspective, it would be interesting to (i) try to formally demonstrate how accurate migration lists are as a representation of human behavior; (ii) study alternative ways of constructing migration lists.
  - Although our MIP model of Section 5.5 solves reasonably quickly given its size (hundreds of thousands of constraints and variables), it may still take a couple of hours to finish running. If the solution time could be reduced to a few minutes, one could think of implementing, and running multiple replications of, a more complex simulation-type model which would solve our MIP model as an intermediate step. This may prove to be a very useful tool in investigating multiple scenarios and/or alternative probability distributions to model customer behavior.

    In Section 5.5.5, we outline a few ideas about how our MIP model can be strengthened with the addition of valid inequalities, but a more comprehensive study of that polytope is necessary.

- When one solves a difficult optimization problem, it is often the case that the model is dynamic but the search strategy is static. Consider the example of branch-and-cut. As the search proceeds, cuts are added to the formulation, which changes the model. Nevertheless, it is usually true that the strategies for node, variable and value selection are fixed before the search starts.

  Ionuţ Aron and I have thought about this issue together and we have found that (i) there are lots of useful branching information that can be collected during the exploration of an enumeration tree (see, for instance, Cornuéjols, Karamanov, and Li, 2004); (ii) SIMPL can be modified to accommodate a variety of "dynamic branching" strategies that use such information.

  Therefore, the challenge here amounts to identifying which type of information is more relevant, how often to collect it, and how to use it to obtain more intelligent and adaptive branching strategies.

1.7 Conclusion

The entire field of optimization has immensely evolved since the early years after World War II. Problems that were once intractable can now be solved in a few minutes on a laptop computer, thanks to numerous theoretical and technological breakthroughs. As a consequence, optimization has become more accessible and, therefore, more popular. Various sectors of our society, such as manufacturing and transportation, rely on optimization tools to survive in the competitive environment of today's economy.

Nevertheless, many important problems are still very hard to solve to optimality. Scheduling (e.g. jobs, sports), routing (e.g. vehicles), and location (e.g. quadratic assignment), are examples of such problems. The recent literature shows examples for which the integration of multiple optimization techniques can bring substantially better results than those obtained with traditional methods alone.

In this thesis we help advance the current knowledge in the area of integrated problem solving. Our integrated modeling and solution system, called SIMPL, is a step toward making integrated optimization methods more easily accessible to a larger group of users. In addition, SIMPL's underlying framework (search-infer-and-relax) is generic enough to effortlessly incorporate a great deal of (if not all) the important results on hybrid methods over the last decade.

SIMPL is still in its early stages of development and it has a long way to go in order to achieve its ambitious goals. We believe, however, that the most important step has been taken and the results obtained so far are encouraging. As described in the previous section, many important questions deserve further investigation. SIMPL's flexibility and expressiveness make it a very welcome research tool that will significantly facilitate new developments in the field of integrated optimization.

Chapter 2

On the Sum Constraint: Relaxation and Applications

Abstract: The global constraint *sum* can be used as a tool to implement summations over sets of variables whose indices are not known in advance. This paper has two major contributions. On the theoretical side, we present the convex hull relaxation for the sum constraint in terms of linear inequalities, whose importance in the context of hybrid models is then justified. On the practical side, we demonstrate the applicability of the sum constraint in a scheduling problem that arises as part of the development of new products in the pharmaceutical and agrochemical industries. This problem can be modeled in two alternative ways: by using the sum constraint in a natural and straightforward manner, or by using the element constraint in a trickier fashion. With the convex hull relaxation developed earlier, we prove that the linear relaxation obtained from the former model is tighter than the one obtained from the latter. Moreover, our computational experiments indicate that the CP model based on the sum constraint is significantly more efficient as well.

2.1 Introduction

The global constraint *sum* can be used as a tool to implement summations over sets of variables whose indices are not known in advance. When studying a scheduling problem in the context of the development of new products in the agrochemical and pharmaceutical industries, we were faced with a subproblem where the sum constraint appears as a natural modeling candidate. We baptize this problem as the *Sequence Dependent Cumulative Cost Problem* (SDCCP), and we describe it in more detail in Sect. 2.2.1. The difficulties faced by previous attempts to solve the original main problem (Honkomp et al., 1997; Jain and Grossmann, 1999) indicate that a combination of Mathematical Programming and Constraint Programming methodologies might produce improved results.

The contributions of this paper are twofold. Firstly, from a hybrid modeling perspective, we give a first step toward better understanding the set of feasible solutions of the sum constraint by presenting its convex hull relaxation in terms of linear inequalities. Building on this result, we look at two alternative formulations of the SDCCP: one using the sum constraint and another using the element constraint. We then show that the former is better than the latter in a well defined sense, i.e. its linear relaxation gives a tighter bound. Secondly, on the purely Constraint Programming side, our computational results for the SDCCP indicate that a model that uses the sum constraint also has the

advantage of being more natural, more concise and computationally more efficient than the alternative model with the element constraint alone.

The remainder of this text is organized as follows. In Sect. 2.2, we describe the semantics of the sum constraint and we present a real-world example of its applicability. In Sect. 2.3, we analyze the importance of having a linear relaxation for the sum constraint, and for any global constraint in general, from the viewpoint of a hybrid modeling paradigm. Given this motivation, in Sect. 2.4 we present the best possible linear relaxation of the sum constraint, i.e. its convex hull. Within the idea of obtaining better dual bounds for optimization problems addressed by a combination of solvers, it then makes sense to assess the relative strengths of relaxations provided by alternative models of a given problem. Section 2.5 presents such a comparison for the SDCCP. Computational results on some randomly generated instances of that problem are given in Sect. 2.6. Finally, Sect. 2.7 summarizes our main conclusions.

2.2 The Sum Constraint and Its Applications

The main purpose of the sum constraint is to implement variable index sets, as defined below.

Definition 1 *Let $S_{j_1}, \ldots, S_{j_d}$ be sets of indices in $\{1, \ldots, n\}$, and let $c_1, \ldots, c_n$ be constants. If y is a variable with domain $D_y = \{j_1, \ldots, j_d\}$, the constraint sum$(y, (S_{j_1}, \ldots, S_{j_d}), (c_1, \ldots, c_n), z)$ states that*

$$z = \sum_{j \in S_y} c_j \ .$$

An extended version of this constraint exists where each constant c_i is replaced by a variable x_i. One can think of the well known element constraint as a special case of the sum constraint in which we have $S_j = \{j\}$ for all $j \in \{1, \ldots, n\}$. Algorithms for achieving hyper arc consistency for the sum constraint with constant terms, and a slightly weaker version of bounds consistency for the case with variable terms are presented in Hooker (2000).

The next section introduces a scheduling problem that turns out to be a natural application of the sum constraint.

2.2.1 The Sequence Dependent Cumulative Cost Problem

Let us define the *Sequence Dependent Cumulative Cost Problem* (SDCCP) as follows. Suppose we are given a set of n tasks that have to be scheduled so that each task is assigned a unique and distinct position in the set $\{1, \ldots, n\}$. Let q_i be the position assigned to task $i \in \{1, \ldots, n\}$, and let p_j contain an arbitrary value associated to the task assigned to position j. We are interested in computing

$$v_i = \sum_{1 \leq j < q_i} p_j \ , \ \forall \ i \in \{1, \ldots, n\} \ . \tag{2.1}$$

Two possible ways of computing v_i are: by using variable subscripts

$$v_i = \sum_{j=1}^{n} p_{\max\{0,\, q_i - q_j\}} \qquad (\text{with } p_0 = 0) \tag{2.2}$$

or by using variable index sets in a more natural way

$$v_i = \sum_{j=1}^{q_i - 1} p_j \ . \tag{2.3}$$

Apart from these constraints, we also have upper and lower bounds on the value of q_i, for every i. In job-shop scheduling terminology, these can be thought of as release dates and due dates. Finally, we are interested in minimizing $\sum_{i=1}^{n} c_i v_i$, where c_i is the cost of performing task i.

This problem appears in the context of scheduling safety/quality tests under resource constraints. It is part of the *New Product Development Problem* (NPDP), which arises in the pharmaceutical and agrochemical industries and has been studied by many authors (Blau et al., 2000; Honkomp et al., 1997; Jain and Grossmann, 1999; Schmidt and Grossmann, 1996). In the NPDP, a given set of products (e.g. newly created drugs) have to pass a series of tests enforced by law before being allowed to reach the market. A stochastic component is present in the sense that each test has a probability of failure. In case one of the tests for a certain product fails, all subsequent tests for that same product are canceled and rescheduling is often necessary in order to make better use of the available resources. The expected cost of test i is given by the probability of executing the test times the cost of performing it. In turn, this probability equals the product of the success probabilities of every test that finishes before test i starts. Another relevant issue is that we usually have technological precedences among the tests, which enforce an initial partial sequencing and contribute to the initial lower and upper bounds on the q_i variables, as mentioned earlier. The objective function of the NPDP seeks to minimize the total cost of performing all the required tests, while satisfying constraints on the number of available laboratories and capacitated personnel. The total cost also takes into account other factors such as a decrease in income due to delays in product commercialization, and the possibility of outsourcing the execution of some tests at a higher price. As reported in Jain and Grossmann (1999), the NPDP is a very hard problem to solve, even when the number of products is small.

Although the objective function of the NPDP turns out to be nonlinear, Jain and Grossmann (1999) show that it can be accurately approximated by a piecewise linear function after using a logarithmic transformation. In this approximation, which we are going to consider here, one needs to calculate, for each test i, an expression of the form (2.1), where p_j equals the logarithm of the success probability of the test assigned to position j. As was done in Jain and Grossmann (1999), an immediate Mixed Integer Programming (MIP) formulation of (2.1) would look like

$$v_i = \sum_{j=1,\, j\neq i}^{n} y_{ji} p_j \ , \ \forall\, i \in \{1, \ldots, n\} \ , \tag{2.4}$$

where $y_{ji} = 1$ if test j precedes test i and $y_{ji} = 0$ otherwise. In this paper, however, we will not explicitly evaluate MIP models for the SDCCP.

2.3 The Logic-Based Modeling Paradigm

The concept of logic-based modeling is far more general and powerful than what is presented in this paper. Hooker (2000) gives a lot of illustrative examples and develops the topic to a large extent. The basic philosophy of this paradigm is very similar to the one behind the Branch-and-Infer (Bockmayr and Kasper, 1998), Mixed Logical/Linear Programming (Hooker and Osorio, 1999) or Branch-and-Check (Thorsteinsson, 2001) frameworks. That is, a problem can be modeled with the Mixed Integer Programming (MIP) language of linear inequalities and also with the more expressive language of Constraint Programming (CP). Then, each type of constraint is handled by its specific solver in a collaborative effort to find an optimal solution to the original problem. That collaboration can be done in a myriad of ways and the best choice will depend on the problem structure.

In real world situations, one can usually identify substructures or smaller parts of a problem that possess nice or well studied properties. These can even be parts of other known problems that, when put together, define the problem under consideration. When trying to write an MIP model, the

modeler is aware of the structure of the problem and how its different parts are connected. However, given the limited expressive power of the vocabulary of linear inequalities, much of this structure will be lost during the translation phase. Many state-of-the-art MIP softwares have algorithms that try to identify some structure inside the linear programs in order to take advantage of them. For example, they can try to use some valid cuts that have been developed for that particular class of problems and improve the quality of the Linear Programming (LP) relaxation, or even apply a different, more specialized, algorithm such as the Network Simplex. But this is not always effective. The more localized view of the problem substructures is usually lost or becomes unrecognizable, and the solver can only work with the global picture of a linear (integer) program. Moreover, a significant portion of the cutting plane theory developed so far is not a default component of current MIP packages.

On the other hand, the Constraint Programming (CP) community has done a lot of work on special purpose algorithms tailored to solve Constraint Satisfaction Problems (Tsang, 1993). These algorithms are encapsulated in the so-called *global constraints* such as *element*, *alldifferent* and *cumulative* (Marriott and Stuckey, 1998). Global constraints can do local inference in the form of constraint propagation, and help to efficiently eliminate infeasible values from the domains of the variables.

Looking at the CP and Operations Research (OR) worlds, it is not hard to see that they can benefit from each other if the local and global views are combined properly. In the logic-based modeling framework, the modeler is able to write the problem formulation with both linear inequalities and more expressive global constraints or logic relations that can better capture and exploit some local structure in the problem. Obviously, this hybrid formulation needs both an LP solver and a constraint solver in order to be useful. But, in principle, this would be transparent to the end-user. In essence, the whole mechanism works as follows. The linear constraints in the logic-based formulation are posted to the LP solver, as usual, and some of them may also be posted to the constraint solver. The constraint solver handles the constraints that cannot be directly posted to the LP solver (e.g. global constraints). When applicable, the continuous relaxation of some global constraints is also sent to the LP solver so as to strengthen the overall relaxation, providing better dual bounds. These linear relaxations of global constraints are sometimes referred to as *dynamic linear relaxations*, because they are supposed to change according to the way the search procedure evolves. This idea also plays a key role behind increasing the performance of hybrid decomposition procedures like Benders decomposition (Benders, 1962) or, more generally, Branch-and-Check, as argued by Thorsteinsson (2001).

It is important to notice that some global constraints have known continuous relaxations that can be added to the set of linear inequalities that are sent to the LP solver. Nevertheless, there are also global constraints for which a continuous relaxation is either not known or too large for practical purposes. In these cases, when building the linear relaxation of the entire logic-based formulation, these global constraints are simply removed and the LP solver ignores their existence. The overall linear relaxation derived from a logic-based formulation is often smaller and weaker than what would be obtained from a pure MIP formulation. On one hand, this allows for a faster solution of the associated linear programs. On the other hand, if we make use of an implicit enumeration algorithm like branch-and-bound, the weaker bounds can result in a larger search tree. Domain reductions resulting from constraint propagation at each node of the tree may help compensate for that.

Given this motivation, the next two sections develop the strongest possible linear relaxation for the sum constraint and theoretically assess its effectiveness when applied to the SDCCP.

2.4 The Convex Hull Relaxation of the Sum Constraint

When dealing with linear optimization problems over disjunctions of polyhedra (Balas, 1998), the convex hull relaxation is the best that one can hope for. More precisely, if Q is the union of a number

of nonempty polyhedra and $\max\{f(x) : x \in Q\}$ is attained at an extreme point x^* of Q, there exists an extreme point x' of the convex hull of Q such that $f(x') = f(x^*)$.

From now on, we assume that $z \geq 0$, $x_i \geq 0$ for all $i \in \{1, \ldots, n\}$, and y is an integer variable whose domain has size $|D_y| = d$. For each $j \in D_y$, $S_j \subseteq \{1, \ldots, n\}$ is a set of indices. We will denote by $\text{conv}(Q)$ the convex hull of an arbitrary set $Q \in \mathbb{R}^m$, for any $m \in \mathbb{N}$.

2.4.1 The Constant Case

For all $i \in \{1, \ldots, n\}$, let c_i be integer constants. Then, the sum constraint

$$\text{sum}(y, (S_{j_1}, \ldots, S_{j_d}), (c_1, \ldots, c_n), z) \tag{2.5}$$

represents the disjunction

$$\bigvee_{j \in D_y} \left(z = \sum_{i \in S_j} c_i \right) \quad . \tag{2.6}$$

Proposition 1 *The convex hull of (2.6) is given by*

$$\min_{j \in D_y} \left\{ \sum_{i \in S_j} c_i \right\} \leq z \leq \max_{j \in D_y} \left\{ \sum_{i \in S_j} c_i \right\} \quad .$$

Proof. For each $j \in D_y$, let $t_j = \sum_{i \in S_j} c_i$. Notice that (2.6) is a one-dimensional problem in which z can take one of $|D_y|$ possible values $t_j \in \mathbb{R}$. The convex hull of these points is simply the line segment connecting all of them. □

2.4.2 The Variable Case

The interesting case is when the list of constants is replaced by a list of variables. This other version of the sum constraint, $\text{sum}(y, (S_{j_1}, \ldots, S_{j_d}), (x_1, \ldots, x_n), z)$, represents the disjunction

$$\bigvee_{j \in D_y} \left(z = \sum_{i \in S_j} x_i \right) \quad . \tag{2.7}$$

Lemma 1 *Let $I = \bigcap_{j \in D_y} S_j$ and, for every $j \in D_y$, let $S'_j = S_j \setminus I$. If $I \neq \emptyset$, (2.7) is the projection of (2.8)–(2.9) onto the space of z and x. Moreover, the convex hull of (2.8)–(2.9) is given by (2.8) and the convex hull of (2.9), together with the non-negativity constraints $z \geq 0$, $x \geq 0$ and $w \geq 0$.*

$$z = \sum_{i \in I} x_i + w \tag{2.8}$$

$$\bigvee_{j \in D_y} \left(w = \sum_{i \in S'_j} x_i \right) \quad . \tag{2.9}$$

Proof. Clearly, for any point $(\overline{z}, \overline{x}) \in \mathbb{R}^{1+n}$ satisfying (2.7), it is easy to find a $\overline{w} \in \mathbb{R}$ such that $(\overline{z}, \overline{x}, \overline{w}) \in \mathbb{R}^{1+n+1}$ satisfies (2.8)–(2.9). For instance, if $(\overline{z}, \overline{x})$ satisfies the r^{th} disjunct in (2.7) (i.e. the one with $j = r$), take $\overline{w} = \sum_{i \in S'_r} x_i$. Conversely, let $(\overline{z}, \overline{x}, \overline{w})$ satisfy (2.8) and the r^{th} disjunct in (2.9). Then, $(\overline{z}, \overline{x})$ satisfies the r^{th} disjunct in (2.7). For the last part, let conv(2.9) denote the convex hull of (2.9). Also, let A be the set of points in $\mathbb{R}^{1+n+1}$ defined by (2.8)–(2.9) and let B be the set of points in $\mathbb{R}^{1+n+1}$ defined by the intersection of (2.8) and conv(2.9). We want to show that $\text{conv}(A) = B$.

$\text{conv}(A) \subseteq B$: Any point in A is built in the following way: pick a point $(\overline{x}, \overline{w})$ that satisfies one of the disjuncts in (2.9) and then set $\overline{z} = \sum_{i \in I} \overline{x}_i + \overline{w}$. Any point $p_3 \in \text{conv}(A)$ can be written as $p_3 = \lambda p_1 + (1-\lambda)p_2$, for some $\lambda \in [0,1]$ and some $p_1 = (\overline{z}_1, \overline{x}^1, \overline{w}_1)$ and $p_2 = (\overline{z}_2, \overline{x}^2, \overline{w}_2)$ from A, built as indicated before. To see that $p_3 = (\overline{z}_3, \overline{x}^3, \overline{w}_3) \in B$, notice that $p_3 \in \text{conv}(2.9)$ because both p_1 and p_2 satisfy (2.9). Finally, p_3 also satisfies (2.8) because $\overline{z}_3 = \lambda \overline{z}_1 + (1-\lambda)\overline{z}_2 = \sum_{i \in I} \overline{x}_i^3 + \overline{w}_3$.

$B \subseteq \text{conv}(A)$: Any point in conv(2.9) can be written as $(\overline{z}, \overline{x}, \overline{w}) = \lambda(\overline{z}_1, \overline{x}^1, \overline{w}_1) + (1-\lambda)(\overline{z}_2, \overline{x}^2, \overline{w}_2)$, where $p_1 = (\overline{z}_1, \overline{x}^1, \overline{w}_1)$ and $p_2 = (\overline{z}_2, \overline{x}^2, \overline{w}_2)$ satisfy some disjuncts in (2.9). When $\overline{z} = \sum_{i \in I} \overline{x}_i + \overline{w}$, then $p = (\overline{z}, \overline{x}, \overline{w}) \in B$. To see that $p \in \text{conv}(A)$, simply notice that p_1 and p_2 can always be chosen with $\overline{z}_1 = \sum_{i \in I} \overline{x}_i^1 + \overline{w}_1$ and $\overline{z}_2 = \sum_{i \in I} \overline{x}_i^2 + \overline{w}_2$, i.e. points in A. □

Before stating the main theorem of this section, we mention an auxiliary result that can be easily proved with standard arguments from Convex Analysis.

Lemma 2 *Let S be an arbitrary set in $\mathbb{R}^{\ell+m}$ and let $\text{Proj}_\ell(S)$ be the projection of S onto the $\mathbb{R}^\ell$ space. Then, $\text{Proj}_\ell(\text{conv}(S)) = \text{conv}(\text{Proj}_\ell(S))$.*

Theorem 1 *Let I and S'_j, for all $j \in D_y$, be defined as in Lemma 1. Let $U = \bigcup_{j \in D_y} S'_j$. The convex hull of (2.7) is given by the projection of (2.8) and (2.10) onto the space of z and x, with $z \geq 0$ and $x \geq 0$.*

$$0 \leq w \leq \sum_{i \in U} x_i \ . \tag{2.10}$$

Proof. By lemmas 1 and 2, it suffices to show that (2.10) is the convex hull of (2.9). Clearly, every point that satisfies (2.9) also satisfies (2.10). To complete the proof, we need to show that any point that satisfies (2.10) is a convex combination of points satisfying (2.9). Let $(\overline{x}, \overline{w})$ satisfy (2.10), and let $K = |U| + 1$. From (2.10), there exists $\alpha \in [0,1]$ such that $\overline{w} = \alpha \sum_{i \in U} \overline{x}_i$. We can write $(\overline{x}, \overline{w})$ as

$$\begin{pmatrix} \overline{x} \\ \overline{w} \end{pmatrix} = \frac{\alpha}{K} \sum_{i \in U} \begin{pmatrix} K\overline{x}_i e^i \\ K\overline{x}_i \end{pmatrix} + \frac{(1-\alpha)}{K} \sum_{i \in U} \begin{pmatrix} K\overline{x}_i e^i \\ 0 \end{pmatrix} + \frac{1}{K} \begin{pmatrix} K\overline{u} \\ 0 \end{pmatrix} \ ,$$

where e^i is the i^{th} unit vector, and $\overline{u}_i = \overline{x}_i$ for every $i \in \{1, \ldots, n\} \setminus U$ and $\overline{u}_i = 0$ otherwise. Notice that every point $(K\overline{x}_i e^i, K\overline{x}_i)$ satisfies the j^{th} disjunct for some j such that $i \in S'_j$. Also, since $\bigcap_{j \in D_y} S'_j = \emptyset$, for every $i \in U$ there exists a disjunct k that is satisfied by the point $(K\overline{x}_i e^i, 0)$. Namely, any k such that $i \notin S'_k$. Finally, $(K\overline{u}, 0)$ trivially satisfies any disjunct in (2.9) by construction. □

After Fourier-Motzkin elimination of w, we get

$$\sum_{i \in I} x_i \leq z \leq \sum_{i \in I \cup U} x_i \ .$$

For the special case when $I = \emptyset$, we have $z = w$ and the previous proof shows that the convex hull of (2.7) is given by (2.10) with w replaced by z, and the non-negativity constraints $z \geq 0$ and $x \geq 0$.

2.5 Comparing Alternative Formulations for the SDCCP

Some parts of the NPDP described in Sect. 2.2.1 present clearly recognizable substructures that could be explored in the context of a logic-based modeling framework. For instance, besides the SDCCP, it is possible to model the influence of the resource limitations on the final schedule of tests by using the

global constraint cumulative in a very natural way. In this paper, however, we will only concentrate on the role of the sum constraint as a tool to model the SDCCP.

Let $p_0 = 0$. We can implement the variable subscripts in (2.2) with (2.11)–(2.13).

$$y_{ij} = q_i - q_j + n, \;\; \forall\, j = 1, \ldots, n, \;\; j \neq i \tag{2.11}$$

$$\text{element}(y_{ij}, [\overbrace{p_0, \ldots, p_0}^{n \text{ times}}, p_1, \ldots, p_{n-1}], z_{ij}), \;\; \forall\, j = 1, \ldots, n, \;\; j \neq i \tag{2.12}$$

$$v_i = \sum_{j=1,\, j \neq i}^{n} z_{ij} \;\; . \tag{2.13}$$

From (2.11), the domain of y_{ij} is $D_{y_{ij}} \subseteq \{1, \ldots, 2n-1\}$. The values between 1 and n represent the situations when $q_i \leq q_j$, which are of no interest. That is why the first n variables in the second argument of (2.12) are set to zero. The variable index sets in (2.3) can be implemented as

$$\text{sum}(q_i, [\{1\}, \{2\}, \{2,3\}, \{2,3,4\}, \ldots, \{2, \ldots, n\}], [p_0, p_1, \ldots, p_{n-1}], v_i) \;\; . \tag{2.14}$$

The next result states that, from the viewpoint of a Linear Programming relaxation, we only need to consider the variable index set formulation (2.14).

Theorem 2 *For each $i \in \{1, \ldots, n\}$, let the initial domain of q_i be $D_i = \{1, \ldots, n\}$. If we impose the constraint alldifferent$(q_1, \ldots, q_n)$, the bounds on v_i given by the relaxation of (2.14) are at least as tight as the bounds given by the relaxation of (2.11)–(2.13).*

To prove this theorem, we need an auxiliary result that follows from the pigeonhole principle.

Lemma 3 *Let $A = \{a_1, \ldots, a_k\} \subseteq \{1, \ldots, n\}$, and let $q_1, \ldots, q_n$ have initial domains $D_1 = \cdots = D_n = \{1, \ldots, n\}$, respectively. When we require the constraint alldifferent$(q_1, \ldots, q_n)$, there exist at least k distinct variables $q_{i_1}, \ldots, q_{i_k}$ such that $a_1 \in D_{i_1}, \ldots, a_k \in D_{i_k}$.*

Proof of Theorem 2. The convex hull relaxation of (2.12) gives $0 \leq z_{ij} \leq \sum_{k \in D_{y_{ij}},\, k \geq n} p_{k-n}$, for all $j = 1, \ldots, n$, $j \neq i$ (see Hooker (2000) for a proof). Therefore, we can write

$$0 \leq v_i \leq \sum_{j=1}^{n} \left(\sum_{k \in D_{y_{ij}},\, k \geq n} p_{k-n} \right) \;\; . \tag{2.15}$$

Let $S_1 = \{0\}$ and $S_j = \{1, \ldots, j-1\}$, for all $j = 2, \ldots, n$. As before, let $U = \bigcup_{j \in D_i} S_j$. By Theorem 1, the convex hull relaxation of (2.14) is given by

$$0 \leq v_i \leq \sum_{k \in U} p_k \;\; . \tag{2.16}$$

We want to show that the RHS of (2.16) is always less than or equal to the RHS of (2.15). We divide the proof in 3 sub-cases.

$q_i = 1$: Clearly, both (2.15) and (2.16) give $v_i = 0$.

$q_i = b > 1$: In this case, the RHS of (2.16) reduces to $\sum_{k=1}^{b-1} p_k$. Notice that, for any j, $D_{y_{ij}}$ will contain values larger than n if and only if D_j contains values smaller than b. But, by Lemma 3, for every number $a \in \{1, \ldots, b-1\}$ there exists at least one variable q_j such that $a \in D_j$. Hence, the RHS of (2.15) reduces to $\sum_{k=1}^{b-1} c_k p_k$, where $c_k \geq 1$.

$|D_i| = d \geq 2$: Let $D_i = \{b_1, \ldots, b_d\}$, with $b_1 < \cdots < b_d$. The RHS of (2.16) reduces to $\sum_{k=1}^{n} c_k p_k$, where $c_k = 1$ if $k \in U$, and $c_k = 0$ otherwise. We will show that the RHS of (2.15) reduces to $\sum_{k=1}^{n} \bar{c}_k p_k$, with $\bar{c}_k \geq c_k$ for every $k = 1, \ldots, n$. Let us start by calculating $\bar{c}_1$, that is, the number of variables q_j for which $n+1 \in D_{y_{ij}}$. Notice that, for every j, $n+1 \in D_{y_{ij}}$ if and only if D_j contains at least one of $b_1 - 1, b_2 - 1, \ldots, b_d - 1$. By Lemma 3, there exist at least d such D_j's ($d-1$ if $b_1 = 1$). Hence, $\bar{c}_1 \geq d$ ($d-1$ if $b_1 = 1$). Analogously, for $k = 2, \ldots, b_1 - 1$, we want to know how many distinct D_j's contain at least one of the numbers $b_1 - k, b_2 - k, \ldots, b_d - k$. This will be equal to the number of distinct $D_{y_{ij}}$'s that contain the value $n + k$. By the same reasoning, we have that $\bar{c}_k \geq d$ whenever $1 \leq k < b_1$. If $b_1 \leq k < b_2$, $b_1 - k \leq 0$ and we need only consider the $d-1$ positive numbers $b_2 - k, b_3 - k, \ldots, b_d - k$. Again, there are at least $d-1$ distinct q_j variables whose domains contain at least one of these numbers. So, $\bar{c}_k \geq d - 1$ for $k = b_1, \ldots, b_2 - 1$. We can now repeat this argument until $k = b_d - 1$. □

2.6 Implementation of the Alternative Models

In order to describe the two CP models for the SDCCP, we will recall some of the notation introduced in Sect. 2.2.1. Let prob(i) denote the probability of success of task $i \in \{1, \ldots, n\}$. Since we have been assuming that all variables are non-negative, the variable p_j will represent $-\log$(prob(task assigned to position j)), for every $j \in \{1, \ldots, n\}$. In this fashion, we can state the SDCCP as a maximization problem with $v_i \geq 0$ for each task i. Let q_i represent the position assigned to task i and let c_i be the cost of performing it. The CP formulation of the SDCCP can be written as

$$\max \sum_{i=1}^{n} c_i v_i \tag{2.17}$$

$$\text{element}(q_i, [p_1, \ldots, p_n], -\log(\text{prob}(i))), \ \ \forall\, i \in \{1, \ldots, n\} \tag{2.18}$$

$$< \text{relationship between } p_i, q_i \text{ and } v_i > \tag{2.19}$$

$$L_i \leq q_i \leq U_i, \ \ \forall\, i \in \{1, \ldots, n\} \ , \tag{2.20}$$

where L_i and U_i are, respectively, lower and upper bounds on the value of q_i.

The two alternative models differ in the way they represent constraints (2.19). In the element constraint model (Model 1), we replace (2.19) by (2.11)–(2.13). In the sum constraint model (Model 2), we replace (2.19) by (2.14), for all $i \in \{1, \ldots, n\}$.

Using the ECL^{i}PS^e (Wallace et al., 1997) constraint logic programming system version 5.3, we implemented an algorithm that achieves a slightly weaker version of bounds consistency for the sum constraint with variable terms, as presented in Hooker (2000). We also implemented a hyper arc consistency algorithm (Hooker, 2000) for the version of the element constraint that indexes a list of variables, as needed for (2.18). The computational times reported in this section are given in CPU seconds of a Sun UltraSPARC-II 360MHz running SunOS 5.7.

2.6.1 Computational Results

To test the performance of the two models described above, we generated random instances of the SDCCP. The size of the instances n, given as the number of tasks, varies from 5 to 10. The values of c_i are randomly picked from a discrete uniform distribution in the interval $[10, 100]$. To control the percentage of tasks with uncertainty in the completion, as well as the percentage of tasks with release dates and due dates, we divided the instances in four classes. Each class is characterized by two numbers a and b that denote, respectively, the percentage of tasks with prob(i) < 1, and

Table 2.1: Comparison of the two CP models for the SDCCP (33%, 33%)

		Model 1		Model 2	
Size	Optimum	Backtracks	Time	Backtracks	Time
5	312.88	2	0.10	2	0.04
6	427.04	14	1.58	7	0.37
7	572.93	67	7.75	36	1.22
8	1,163.76	149	79.10	63	8.18
9	1,183.07	1,304	518.38	328	44.33
10	1,102.39	31,152	6,293.25	4,648	308.60

Table 2.2: Comparison of the two CP models for the SDCCP (50%, 33%)

		Model 1		Model 2	
Size	Optimum	Backtracks	Time	Backtracks	Time
5	180.39	4	0.15	3	0.05
6	367.76	18	1.07	10	0.24
7	898.01	41	6.25	24	1.06
8	1,435.72	303	52.47	91	4.96
9	1,595.03	1,803	187.27	881	19.41
10	1,505.62	67,272	8,605.18	15,345	561.90

the percentage of tasks with non-trivial release dates and due dates. The (a, b) pairs chosen for our experiments are: (33%, 33%), (50%, 33%), (50%, 50%) and (75%, 50%). The rationale behind this choice is to try to evaluate the behavior of the models under different problem configurations. For example, for problems of size 6 in class (50%, 33%), 50% of the tasks have prob(i) chosen randomly from a uniform distribution in the interval $[0, 1]$. The other half of the prob(i) values are set to 1. Also, for 33% of the tasks, L_i and U_i are randomly chosen from a discrete uniform distribution in the interval $[1, 6]$. For the remaining 67% of the tasks, $L_i = 1$ and $U_i = 6$.

For each one of the four classes of instances, the numbers reported in tables 2.1 through 2.4 are the average values over 10 different runs for each instance size ranging from 5 to 9, and average values over 3 runs for instances of size 10.

Our results indicate that, as the instance size increases, Model 1, which is based on the element constraint, requires a significantly larger number of backtracks than Model 2, which uses the sum constraint. In terms of computational time, solving Model 2 tends to be roughly one order of magnitude faster than solving Model 1. One reason for this behavior can be attributed to the fact that Model 2, besides being more compact, also provides more efficient pruning, since Model 1 is essentially simulating the sum constraint with a number of element constraints.

2.7 Conclusions

We study a scheduling problem that appears as a substructure of a real-world problem in the agrochemical and pharmaceutical industries called the New Product Development Problem (NPDP) (Honkomp et al., 1997; Jain and Grossmann, 1999). We refer to that subproblem as the Sequence Dependent

Table 2.3: Comparison of the two CP models for the SDCCP (50%, 50%)

		Model 1		Model 2	
Size	Optimum	Backtracks	Time	Backtracks	Time
5	330.01	3	0.13	3	0.04
6	444.10	7	0.67	5	0.17
7	995.88	28	6.44	14	0.94
8	1,488.54	121	68.04	48	6.28
9	995.72	1,216	293.11	358	31.74
10	2,207.05	973	524.69	121	21.15

Table 2.4: Comparison of the two CP models for the SDCCP (75%, 50%)

		Model 1		Model 2	
Size	Optimum	Backtracks	Time	Backtracks	Time
5	670.65	5	0.20	4	0.05
6	825.83	8	0.47	4	0.10
7	1,241.32	46	4.05	23	0.59
8	1,713.76	96	24.19	46	3.14
9	2,346.90	1,943	233.54	869	19.86
10	2,512.04	4,025	1,794.19	1,311	155.76

Cumulative Cost Problem (SDCCP).

Given the description of the SDCCP, the sum constraint is identified as a natural tool, simplifying the modeling effort and making it more intuitive. Apart from these advantages, our computational experiments show that the sum constraint exhibits a significantly improved performance over an alternative model that uses the element constraint. From a more theoretical point of view, we study the characteristics of the set of feasible solutions of the sum constraint. We provide the representation of the convex hull of this set of feasible solutions and we prove that this linear relaxation gives tighter bounds than the linear relaxation of the aforementioned alternative model. The relevance of this result is related to hybrid modeling efforts. One promising idea is to use linear relaxations of global constraints to help in the process of solving optimization problems through a combination of solvers. These linear relaxations, or at least some of their valid inequalities, may contribute to speed up the solution process by improving the bounds on the optimal solution value.

As an extension of this work, one can try to look at different contexts in which the sum constraint also fits well as a modeling device, and then compare its performance against other modeling possibilities. Finally, the impact of modeling the SDCCP with the sum constraint while solving the more general NPDP is another topic that deserves further investigation.

Chapter 3

SIMPL: A System for Integrating Optimization Techniques

Joint work with: Ionuţ Aron and John Hooker.

Abstract: In recent years, the Constraint Programming (CP) and Operations Research (OR) communities have explored the advantages of combining CP and OR techniques to formulate and solve combinatorial optimization problems. These advantages include a more versatile modeling framework and the ability to combine complementary strengths of the two solution technologies. This research has reached a stage at which further development would benefit from a general-purpose modeling and solution system. We introduce here a system for integrated modeling and solution called SIMPL. Our approach is to view CP and OR techniques as special cases of a single method rather than as separate methods to be combined. This overarching method consists of an infer-relax-restrict cycle in which CP and OR techniques may interact at any stage. We describe the main features of SIMPL and illustrate its usage with examples.

3.1 Introduction

In recent years, the Constraint Programming (CP) and Operations Research (OR) communities have explored the advantages of combining CP and OR techniques to formulate and solve combinatorial optimization problems. These advantages include a more versatile modeling framework and the ability to combine complementary strengths of the two solution technologies. Examples of existing programming languages that provide mechanisms for combining CP and OR techniques are ECL^{i}PS^e (Rodošek et al., 1999; Wallace et al., 1997), OPL (Van Hentenryck et al., 1999) and Mosel (Colombani and Heipcke, 2004).

Hybrid methods tend to be most effective when CP and OR techniques interact closely at the micro level throughout the search process. To achieve this one must often write special-purpose code, which slows research and discourages broader application of integrated methods. We address this situation by introducing here a system for integrated modeling and solution called SIMPL (Modeling Language for Integrated Problem Solving). The SIMPL modeling language formulates problems in such a way as to reveal problem structure to the solver. The solver executes a search algorithm that invokes CP and OR techniques as needed, based on problem characteristics.

The design of such a system presents a significant research problem in itself, since it must be flexible enough to accommodate a wide range of integration methods and yet structured enough to allow high-level implementation of specific applications. Our approach, which is based partly on a proposal in Hooker (2000, 2003), is to view CP and OR techniques as special cases of a single method rather than as separate methods to be combined. This overarching method consists of an infer-relax-restrict cycle in which CP and OR techniques may interact at any stage.

This paper is organized as follows. In Sect. 3.2, we briefly review some of the fundamental ideas related to the combination of CP and OR that are relevant to the development of SIMPL. We describe the main concepts behind SIMPL in Sect. 3.3 and talk about implementation details in Sect. 3.4. Section 3.5 presents a few examples of how to model optimization problems in SIMPL, explaining the syntax and semantics of the language. Finally, Sect. 3.6 outlines some additional features provided by SIMPL, and Sect. 3.7 discusses directions for future work.

3.2 Previous Work

A comprehensive survey of the literature on the cooperation of logic-based, Constraint Programming (CP) and Operations Research (OR) methods can be found in Hooker (2002). Some of the concepts that are most relevant to the work presented here are: decomposition approaches (e.g. Benders (1962)) that solve parts of the problem with different techniques (Eremin and Wallace, 2001; Hooker, 2000; Hooker and Ottosson, 2003; Hooker and Yan, 1995; Jain and Grossmann, 2001; Thorsteinsson, 2001); allowing different models/solvers to exchange information (Rodošek et al., 1999); using linear programming to reduce the domains of variables or to fix them to certain values (Beringer and de Backer, 1995; Focacci et al., 1999; Rodošek et al., 1999); automatic reformulation of global constraints as systems of linear inequalities (Refalo, 2000); continuous relaxations of global constraints and disjunctions of linear systems (Balas, 1998; Hooker, 2000; Hooker and Osorio, 1999; Hooker and Yan, 2002; Ottosson et al., 1999; Williams and Yan, 2001; Yan and Hooker, 1999; Yunes, 2002); understanding the generation of cutting planes as a form of logical inference (Bockmayr and Eisenbrand, 2000; Bockmayr and Kasper, 1998); strengthening the problem formulation by embedding the generation of valid cutting planes into CP constraints (Focacci et al., 2000); maintaining the continuous relaxation of a constraint updated when the domains of its variables change (Refalo, 1999); and using global constraints as a key component in the intersection of CP and OR (Milano et al., 2002).

Ideally, one would like to incorporate all of the above techniques into a single modeling and solving environment, in a clean and generic way. Additionally, this environment should be flexible enough to accommodate improvements and modifications with as little extra work as possible. In the next sections, we present the concepts behind SIMPL that aim at achieving those objectives.

3.3 SIMPL Concepts

We first review the underlying solution algorithm and then indicate how the problem formulation helps to determine how particular problems are solved.

3.3.1 The Solver

SIMPL solves problems by enumerating problem restrictions. (A restriction is the result of adding constraints to the problem.) Each node of a classical branch-and-bound tree, for example, can be viewed as a problem restriction defined by fixing certain variables or reducing their domains. Local search methods fit into the same scheme, since they examine a sequence of neighborhoods, each of

which is the feasible set of a problem restriction. Thus SIMPL implements both exact and heuristic methods within the same architecture.

The search proceeds by looping through an infer-relax-restrict cycle: it infers new constraints from the current problem restriction, then formulates and solves relaxations of the augmented problem restriction, and finally moves to another problem restriction to be processed in the same way. The user specifies the overall search procedure from a number of options, such as depth-first branching, local search, or Benders decomposition. The stages in greater detail are as follows.

Infer. New constraints are deduced from the original ones and added to the current problem restriction. For instance, a filtering algorithm can be viewed as inferring indomain constraints that reduce the size of variable domains. A cutting plane algorithm can generate inequality constraints that tighten the continuous relaxation of the problem as well as enhance interval propagation.

Relax. One or more relaxations of the current problem restriction are formulated and solved by specialized solvers. For instance, continuous relaxations of some or all of the constraints can be collected to form a relaxation of the entire problem, which is solved by a linear or nonlinear programming subroutine. The role of relaxations is to help direct the search, as described in the next step.

Restrict. The relaxations provide information that dictates which new restrictions are generated before moving to the next restriction. In a tree search, for example, SIMPL creates new restrictions by branching on a constraint that is violated by the solution of the current relaxation. If several constraints are violated, one is selected according to user- or system-specified priorities (see Sect. 3.4.3). Relaxations can also influence which restriction is processed next, for instance by providing a bound that prunes a branch-and-bound tree.

If desired, an inner infer-relax loop can be executed repeatedly before moving to the next problem restriction, since the solution of the relaxation may indicate further useful inferences that can be drawn (post-relaxation inference). An example would be separating cuts, which are cutting planes that "cut off" the solution of the relaxation (see Sect. 3.4.3).

The best-known classical solution methods are special cases of the infer-relax-restrict procedure:

- In a typical *CP solver*, the inference stage consists primarily of domain reduction. The relaxation stage builds a (weak) relaxation simply by collecting the reduced domains into a constraint store. New problem restrictions are created by splitting a domain in the relaxation.

- In a *branch-and-bound solver* for integer programming, the inference stage can be viewed as "preprocessing" that takes place at the root node and possibly at subsequent nodes. The relaxation stage drops the integrality constraints and solves the resulting problem with a linear or perhaps nonlinear programming solver. New problem restrictions are created by branching on an integrality constraint; that is, by branching on a variable with a fractional value in the solution of the relaxation.

- A *local search* procedure typically chooses the next solution to be examined from a neighborhood of the current solution. Thus local search can be regarded as enumerating a sequence of problem restrictions, since each neighborhood is the feasible set of a problem restriction. The "relaxation" of the problem restriction is normally the problem restriction itself, but need not be. The restriction may be solved to optimality by an exhaustive search of the neighborhood, as in tabu search (where the tabu list is part of the restriction). Alternatively, a suboptimal solution may suffice, as in simulated annealing, which selects a random element of the neighborhood.

- In *Branch-and-Infer* (Bockmayr and Kasper, 1998), the relaxation stage is not present and branching corresponds to creating new problem restrictions.

An important advantage of SIMPL is that it can create new infer-relax-restrict procedures that suit the problem at hand. One example is a hybrid algorithm, introduced in Hooker (2000); Hooker and Yan (1995), that is obtained through a generalization of Benders decomposition. It has provided some of the most impressive speedups achieved by hybrid methods (Eremin and Wallace, 2001; Hooker, 2003, 2004; Hooker and Ottosson, 2003; Jain and Grossmann, 2001). A Benders algorithm distinguishes a set of primary variables that, when fixed, result in an easily-solved subproblem. Solution of an "inference dual" of the subproblem yields a Benders cut, which is added to a master problem containing only the primary variables. Solution of the master problem fixes the primary variables to another value, and the process continues until the optimal values of the master problem and subproblem converge. In typical applications, the master problem is an integer programming problem and the subproblem a CP problem. This method fits nicely into the infer-relax-restrict paradigm, since the subproblems are problem restrictions and master problems are relaxations. The solution of the relaxation guides the search by defining the next subproblem.

The choice of constraints in a SIMPL model can result in novel combinations of CP, OR and other techniques. This is accomplished as described in Sect. 3.3.2.

3.3.2 Modeling

SIMPL is designed so that the problem formulation itself determines to a large extent how CP, OR, and other techniques interact. The basic idea is to view each constraint as invoking specialized procedures that exploit the structure of that particular constraint. Since some of these procedures may be from CP and some from OR, the two approaches interact in a manner that is dictated by which constraints appear in the problem.

This idea of associating constraints with procedures already serves as a powerful device for exploiting problem substructure in CP, where a constraint typically activates a specialized filtering algorithm. SIMPL extends the idea by associating each constraint with procedures in all three stages of the search. Each constraint can (a) activate inference procedures, (b) contribute constraints to one or more relaxations, and (c) generate further problem restrictions if the search branches on that particular constraint.

If a group of constraints exhibit a common structure—such as a set of linear inequalities, flow balance equations, or logical formulas in conjunctive normal form—they are identified as such so that the solver can take advantage of their structure. For instance, a resolution-based inference procedure might be applied to the logical formulas.

The existing CP literature typically provides inference procedures (filters) only for CP-style global constraints, and the OR literature provides relaxations (cutting planes) only for structured groups of linear inequalities. This poses the research problem of finding specialized relaxations for global constraints and specialized filters for structured linear systems. Some initial results along this line are surveyed in Hooker (2002).

Some examples should clarify these ideas. The global constraint *element* is important for implementing variable indices. Conventional CP solvers associate *element* with a specialized filtering algorithm, but useful linear relaxations, based on OR-style polyhedral analysis, have recently been proposed as well (Hooker et al., 1999). Thus each *element* constraint can activate a domain reduction algorithm in the inference stage and generate linear inequalities, for addition to a continuous relaxation, in the relaxation stage. If the search branches on a violated *element* constraint, then

new problem restrictions are generated in a way that makes sense when that particular constraint is violated.

The popular *all-different* and *cumulative* constraints are similar in that they also have well-known filters (Régin, 1994; Baptiste et al., 2001) and were recently provided with linear relaxations (Williams and Yan, 2001; Hooker and Yan, 2002). These relaxations are somewhat weak and may not be useful, but the user always has the option of turning off or on the available filters and relaxations, perhaps depending on the current depth in the search tree.

Extensive polyhedral analysis of the traveling salesman problem in the OR literature (Gutin and Punnen, 2002; Lawler et al., 1985) provides an effective linear relaxation of the *cycle* constraint. In fact, SIMPL has the potential to make better use of the traditional OR literature than commercial OR solvers. Structured groups of inequalities can be represented by global constraints that trigger the generation of specialized cutting planes, many of which go unused in today's general-purpose solvers.

3.4 From Concepts to Implementation

SIMPL is implemented in C++ as a collection of object classes, as shown in Fig. 3.1. This makes it

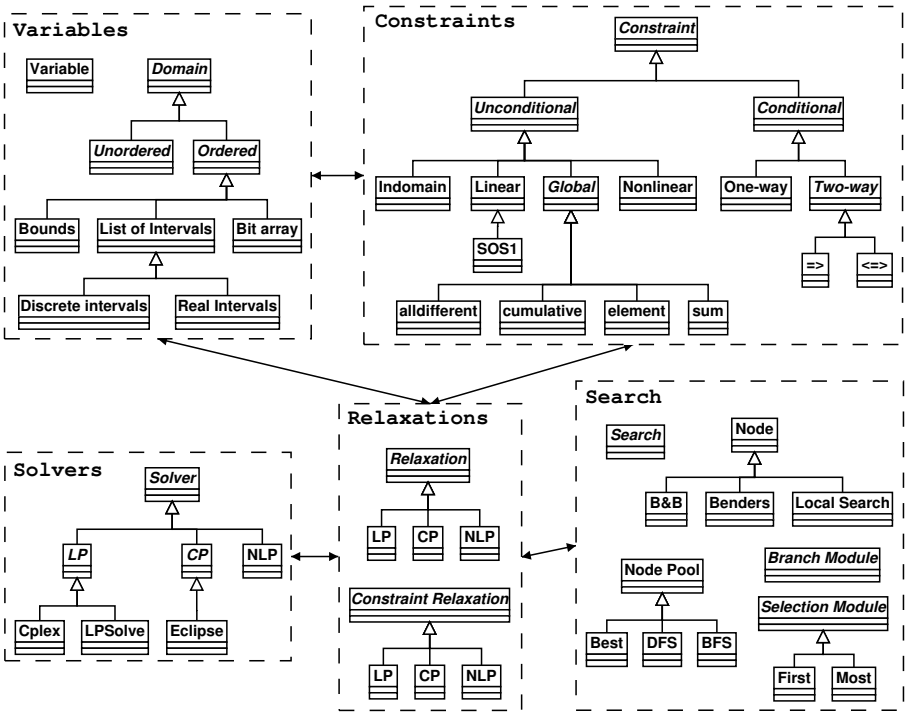

Figure 3.1: Main components of SIMPL

easy to add new components to the system by making only localized changes that are transparent to the other components. Examples of components that can be included are: new constraints, different relaxations for existing constraints, new solvers, improved inference algorithms, new branching modules and selection modules, alternative representations of domains of variables, etc. The next sections describe some of these components in detail.

3.4.1 Multiple Problem Relaxations

Each iteration in the solution of an optimization problem P examines a *restriction* N of P. In a tree search, for example, N is the problem restriction at the current node of the tree. Since solving N can be hard, we usually solve a *relaxation*[1] N_R of N, or possibly several relaxations.

In an integrated CP-IP modeling system, the linear constraints in the hybrid formulation are posted to a Linear Programming (LP) solver, and some (or all) of them may be posted to a CP solver as well. The CP solver also handles the constraints that cannot be directly posted to the LP solver (e.g. global constraints). Notice that each solver only deals with a relaxation of the original problem P (i.e. a subset of its constraints). In this example, each problem restriction N has two relaxations: an *LP relaxation* and a *CP relaxation*. Extending this idea to more than two kinds of relaxations is straightforward.

In principle, the LP relaxation of N could simply ignore the constraints that are not linear. Nevertheless, this relaxation can be strengthened by the addition of linear relaxations of those constraints, if available (see Sect. 3.4.2).

3.4.2 Constraints and Constraint Relaxations

In SIMPL, the actual representation of a constraint of the problem inside any given relaxation is called a *constraint relaxation*. Every constraint can be associated with a list of constraint relaxation objects, which specify the relaxations of that constraint that will be used in the solution of the problem under consideration. To *post* a constraint means to add its constraint relaxations to all the appropriate problem relaxations. For example, both the LP and the CP relaxations of a linear constraint are equal to the constraint itself. The CP relaxation of the *element* constraint is clearly equal to itself, but its LP relaxation can be the convex hull formulation of its set of feasible solutions (Hooker, 2000). Besides the ones already mentioned in Sect. 3.3.2, other constraints for which linear relaxations are known include *cardinality rules* (Yan and Hooker, 1999) and *sum* (Yunes, 2002).

For a branch-and-bound type of search, the problem relaxations to be solved at a node of the enumeration tree depend on the state of the search at that node. In theory, at every node, the relaxations are to be created from scratch because constraint relaxations are a function of the domains of the variables of the original (non-relaxed) constraint. Nevertheless, this can be very inefficient because a significant part of the constraints in the relaxations will be the same from node to node. Hence, we divide constraint relaxations in two types:

Static: those that change very little (in structure) when the domains of its variables change (e.g. relaxations of linear constraints are equal to themselves, perhaps with some variables removed due to fixing);

Volatile: those that radically change when variable domains change (e.g. some linear relaxations of global constraints).

To update the problem relaxations when we move from one node in the search tree to another, it suffices to recompute volatile constraint relaxations only. This kind of update is not necessary for the purpose of creating valid relaxations, but it is clearly beneficial from the viewpoint of obtaining stronger bounds.

[1] In general, we say that problem Q_R is a relaxation of problem Q if the feasible region of Q_R contains the feasible region of Q.

3.4.3 Search

The main search loop in SIMPL is implemented as shown in Fig. 3.2. Here, N is again the current prob-

```
procedure Search(A)
    If A ≠ ∅ and stopping criteria not met
       N := A.getNextNode()
       N.explore()
       A.addNodes(N.generateRestrictions())
       Search(A)
```

Figure 3.2: The main search loop in SIMPL

lem restriction, and A is the current list of restrictions waiting to be processed. Depending on how A, N and their subroutines are defined, we can have different types of search, as mentioned in Sect. 3.3.1. The routine N`.explore()` implements the infer-relax sequence. The routine N`.generateRestrictions()` creates new restrictions, and A`.addNodes()` adds them to A. Routine A`.getNextNode()` implements a mechanism for selecting the next restriction, such as depth-first, breadth-first or best bound.

In tree search, N is the problem restriction that corresponds to the current node, and A is the set of open nodes. In local search, N is the restriction that defines the current neighborhood, and A is the singleton containing the restriction that defines the next neighborhood to be searched. In Benders decomposition, N is the current subproblem and A is the singleton containing the next subproblem to be solved. In the case of Benders, the role of N`.explore()` is to infer Benders cuts from the current subproblem, add them to the master problem, and solve the master problem. N`.generateRestrictions()` uses the solution of the master problem to create the next subproblem.

In the sequel, we will restrict our attention to branch-and-bound search.

Node Exploration.

Figure 3.3 describes the behavior of N`.explore()` for a branch-and-bound type of search. Steps 1 and

```
1.  Pre-relaxation inference
2.  Repeat
3.      Solve relaxations
4.      Post-relaxation inference
5.  Until (no changes) or (iteration limit)
```

Figure 3.3: The node exploration loop in branch-and-bound

4 are inference steps where we try to use the information from each relaxation present in the model to the most profitable extent. Section 3.4.4 provides further details about the types of inference used in those steps. The whole loop can be repeated multiple times, as long as domains of variables keep changing because of step 4, and the maximum number of iterations has not been reached. This process of re-solving relaxations and looking for further inferences behaves similarly to a fix point calculation.

Branching.

SIMPL implements a tree search by branching on constraints. This scheme is considerably more powerful and generic than branching on variables alone. If branching is needed, it is because some constraint of the problem is violated and that constraint should "know" what to do. This knowledge is embedded in the so called *branching module* of that constraint. For example, if a variable $x \in \{0, 1\}$ has a fractional value in the current LP, its indomain constraint I_x is violated. The branching module

of I_x will then output two constraints: $x \in \{0\}$ and $x \in \{1\}$, meaning that two subproblems should be created by the inclusion of those two new constraints. In this sense, branching on the variable x can be interpreted as branching on I_x. In general, a branching module returns a sequence of *sets* of constraints $C_1, \ldots, C_k$. This sequence means that k subproblems should be created, and subproblem i can be constructed from the current problem by the inclusion of all constraints present in the set C_i. There is no restriction on the types of constraints that can be part of the sets C_i.

Clearly, there may be more than one constraint violated by the solution of the current set of problem relaxations. A *selection module* is the entity responsible for selecting, from a given set of constraints, the one on which to branch next. Some possible criteria for selection are picking the first constraint found to be violated or the one with the largest degree of violation.

3.4.4 Inference

We now take a closer look at the inference steps of the node exploration loop in Fig. 3.3. In step 1 (pre-relaxation inference), one may have domain reductions or the generation of new implied constraints (see Hooker and Osorio (1999)), which may have been triggered by the latest branching decisions. If the model includes a set of propositional logic formulas, this step can also execute some form of resolution algorithm to infer new resolvents. In step 4 (post-relaxation inference), other types of inference may take place, such as fixing variables by reduced cost or the generation of cutting planes. After that, it is possible to implement some kind of primal heuristic or to try extending the current solution to a feasible solution in a more formal way, as advocated in Sect. 9.1.3 of Hooker (2000).

Since post-relaxation domain reductions are associated with particular relaxations, the reduced domains that result are likely to differ across relaxations. Therefore, at the end of the inference steps, a synchronization step must be executed to propagate domain reductions across different relaxations. This is shown in Fig. 3.4. In step 6, D_v^r denotes the domain of v inside relaxation r, and D_v works as a

```
1.  V := ∅
2.  For each problem relaxation r
3.      V_r := variables with changed domains in r
4.      V := V ∪ V_r
5.      For each v ∈ V_r
6.          D_v := D_v ∩ D_v^r
7.  For each v ∈ V
8.      Post constraint v ∈ D_v
```

Figure 3.4: Synchronizing domains of variables across multiple relaxations

temporary domain for variable v, where changes are centralized. The initial value of D_v is the current domain of variable v. By implementing the changes in the domains via the addition of indomain constraints (step 8), those changes will be transparently undone when the search moves to a different part of the enumeration tree. Similarly, those changes are guaranteed to be redone if the search returns to descendents of the current node at a later stage.

3.5 SIMPL Examples

SIMPL's syntax is inspired by OPL (Van Hentenryck et al., 1999), but it includes many new features.

Apart from the resolution algorithm used in Sect. 3.5.3, SIMPL is currently able to run all the examples presented in this section. Problem descriptions and formulations were taken from Chapter 2 of Hooker (2000).

3.5.1 A Hybrid Knapsack Problem

Let us consider the following integer knapsack problem with a side constraint.

$$\begin{array}{rl} \min & 5x_1 + 8x_2 + 4x_3 \\ \text{subject to} & 3x_1 + 5x_2 + 2x_3 \geq 30 \\ & \textit{all-different}(x_1, x_2, x_3) \\ & x_j \in \{1, 2, 3, 4\}, \text{ for all } j \end{array}$$

To handle the *all-different* constraint, a pure MIP model would need auxiliary binary variables: $y_{ij} = 1$ if and only if $x_i = j$. A SIMPL model for the above problem is shown in Fig. 3.5. The model starts with a DECLARATIONS section in which constants and variables are defined. The objective function is defined in line 06. Notice that the range over which the index `i` takes its values need not be explicitly stated. In the CONSTRAINTS section, the two constraints of the problem are named `totweight` and `distinct`, and their definitions show up in lines 09 and 12, respectively. The RELAXATION statements in lines 10 and 13 indicate the relaxations to which those constraints should be posted. The linear constraint will be present in both the LP and the CP relaxations, whereas the `alldiff` constraint will only be present in the CP relaxation. In the SEARCH section, line 15 indicates we will do branch-and-bound (BB) with depth-first search (DEPTH). The BRANCHING statement in line 16 says that we will branch on the first of the `x` variables that is not integer (remember from Sect. 3.4.3 that branching on a variable means branching on its indomain constraint).

```
01.  DECLARATIONS
02.      n = 3; cost[1..n] = [5,8,4]; weight[1..n] = [3,5,2]; limit = 30;
03.      DISCRETE RANGE xRange = 1 TO 4;
04.      x[1..n] IN xRange;
05.  OBJECTIVE
06.      MIN SUM i OF cost[i]*x[i]
07.  CONSTRAINTS
08.      totweight MEANS {
09.         SUM i OF weight[i]*x[i] >= limit
10.         RELAXATION = {LP, CS} }
11.      distinct MEANS {
12.         alldiff(x)
13.         RELAXATION = {CS} }
14.  SEARCH
15.      TYPE = {BB:DEPTH}
16.      BRANCHING = {x:FIRST}
```

Figure 3.5: SIMPL model for the Hybrid Knapsack Problem

Initially, bounds consistency maintenance in the CP solver removes value 1 from the domain of x_2 and the solution of the LP relaxation is $x = (2\frac{2}{3}, 4, 1)$. After branching on $x_1 \leq 2$, bounds consistency determines that $x_1 \geq 2$, $x_2 \geq 4$ and $x_3 \geq 2$. At this point, the `alldiff` constraint produces further domain reduction, yielding the feasible solution $(2, 4, 3)$. Notice that no LP relaxation had to be solved at this node. In a similar fashion, the CP solver may be able to detect infeasibility even before the linear relaxation has to be solved.

3.5.2 A Lot Sizing Problem

A total of P products must be manufactured over T days on a single machine of limited capacity C, at most one product each day. When manufacture of a given product i begins, it may proceed for several days, and there is a minimum run length r_i. Given a demand d_{it} for each product i on each day t, it is usually necessary to hold part of the production of a day for later use, at a unit cost of

h_{it}. Changing from product i to product j implies a setup cost q_{ij}. Frequent changeovers allow for less holding cost but incur more setup cost. The objective is to minimize the sum of the two types of costs while satisfying demand.

Let $y_t = i$ if and only if product i is chosen to be produced on day t, and let x_{it} be the quantity of product i produced on day t. In addition, let u_t, v_t and s_{it} represent, respectively, for day t, the holding cost, the changeover cost and the ending stock of product i. Figure 3.6 exhibits a SIMPL model for this problem. We have omitted the data that initializes matrices d, h, q and r. We have also left out the statements that set $y_0 = 0$ and $s_{i0} = 0$ for $i \in \{1, \ldots, P\}$.

```
01.  DECLARATIONS
02.      P = 5; T = 10; C = 50;
03.      d[1..P,1..T] = ; h[1..P,1..T] = ; q[0..P,1..P] = ; r[1..P] = ;
04.      CONTINUOUS RANGE xRange = 0 TO C;
05.      DISCRETE RANGE yRange = 0 TO P;
06.      x[1..P,1..T] IN xRange; y[0..T] IN yRange;
07.      u[1..T], v[1..T], s[1..P,0..T] IN nonegative;
08.  OBJECTIVE
09.      MIN SUM t OF u[t] + v[t]
10.  RELAXATIONS
11.      LP, CS
12.  CONSTRAINTS
13.      holding MEANS { u[t] >= SUM i OF h[i,t]*s[i,t] FORALL t }
14.      setup MEANS { v[t] >= q[y[t-1],y[t]] FORALL t }
15.      stock MEANS { s[i,t-1] + x[i,t] = d[i,t] + s[i,t] FORALL i,t }
16.      linkyx MEANS { y[t] <> i -> x[i,t] = 0 FORALL i,t }
17.      minrun MEANS {
18.         y[t-1] <> i and y[t] = i ->
19.         (y[t+k] = i FORALL k IN 1 TO r[i]-1) FORALL i,t }
20.  SEARCH
21.      TYPE = {BB:BEST}
22.      BRANCHING = {setup:MOST}
```

Figure 3.6: SIMPL model for the Lot Sizing Problem

In line 07, we use the predefined continuous range `nonegative`. Notice the presence of a new section called RELAXATIONS, whose role in this example is to define the default relaxations to be used. As a consequence, the absence of RELAXATION statements in the declaration of constraints means that all constraints will be posted to both the LP and CS relaxations. The `holding` and `stock` constraints define, respectively, holding costs and stock levels in the usual way. The `setup` constrains make use of variable indexing to obtain the desired meaning for the v_t variables. The CS relaxation of these constraints uses *element* constraints, and the LP relaxation uses the corresponding linear relaxation of *element*. The symbol `->` in lines 16 and 18 implements a one-way link constraint of the form $A \rightarrow B$ (Hooker and Osorio, 1999). This means that whenever condition A is true, B is imposed as a constraint of the model, but we do not worry about the contrapositive. Condition A may be a more complicated logical statement and B can be any collection of arbitrary constraints. There are also two-way link constraints such as "implies" (`=>`) and "if and only if" (`<=>`) available in SIMPL. Here, the `linkyx` constraints ensure that x_{it} can only be positive if $y_t = i$, and the `minrun` constraints make production last the required minimum length. The statements in lines 21 and 22 define a branch-and-bound search with best-bound node selection, and branching on the most violated of the `setup` constraints, respectively.

3.5.3 Processing Network Design

This problem consists of designing a chemical processing network. In practice one usually starts with a network that contains all the processing units and connecting links that could eventually be built

(i.e. a superstructure). The goal is to select a subset of units that deliver the required outputs while minimizing installation and processing costs. The discrete element of the problem is the choice of units, and the continuous element comes in determining the volume of flow between units. Let us consider the simplified superstructure in Fig. 3.7(a). Unit 1 receives raw material, and units 4, 5 and

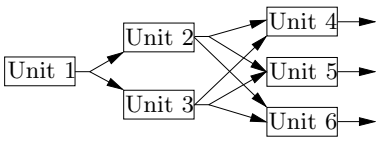

Figure 3.7: (a) Network superstructure

```
build MEANS {
   y1 => (y2 or y3), y3 => y4, y2 => y1,
   y3 => (y5 or y6), y2 => (y4 or y5),
   y4 => (y2 or y3), y2 => y6, y3 => y1,
   y5 => (y2 or y3), y6 => (y2 or y3)
   RELAXATION = {LP, CS}
   INFERENCE = {RESOLUTION} }
```

(b) The INFERENCE statement in SIMPL

6 generate finished products. The output of unit 1 is then processed by unit 2 and/or 3, and their outputs undergo further processing. For the purposes of this example, we will concentrate on the selection of units, which is amenable to the following type of logical reasoning. Let the propositional variable y_i be true when unit i is installed and false otherwise. From Fig. 3.7(a), it is clearly useless to install unit 1 unless one installs unit 2 or unit 3. This condition can be written as $y_1 \Rightarrow (y_2 \vee y_3)$. Other rules of this kind can be derived in a similar way. SIMPL can take advantage of the presence of such rules in three ways: it can relax logical propositions into linear constraints; it can use the propositions *individually* as two-way link constraints (see Sect. 3.5.2); and it can use the propositions *collectively* with an inference algorithm to deduce stronger facts. The piece of code in Fig. 3.7(b) shows how one would group this collection of logical propositions as a constraint in SIMPL. In addition to the known RELAXATION statement, this example introduces an INFERENCE statement whose role is to attach an inference algorithm (resolution) to the given group of constraints. This algorithm will be invoked in the pre-relaxation inference step, as described in Sect. 3.4.4. Newly inferred resolvents can be added to the problem relaxations and may help the solution process.

3.5.4 Benders Decomposition

Recall from Sect. 3.4.3 that Benders decomposition is a special case of SIMPL's search mechanism. Syntatically, to implement Benders decomposition the user only needs to include the keyword MASTER in the RELAXATION statement of each constraint that is meant to be part of the master problem (remaining constraints go to the subproblem), and declare TYPE = {BENDERS} in the SEARCH section. As is done for linear relaxations of global constraints, Benders cuts are generated by an algorithm that resides inside each individual constraint. At present, we are in the process of implementing the class *Benders* in the diagram of Fig. 3.1.

3.6 Other SIMPL Features

Supported Solvers.

Currently, SIMPL can interface with CPLEX (ILOG S. A.) and LP_SOLVE (Berkelaar) as LP solvers, and with ECL^{i}PS^e (Wallace et al., 1997) as a CP solver. Adding a new solver to SIMPL is an easy task and amounts to implementing an interface to that solver's callable library, as usual. The rest of the system does not need to be changed or recompiled. One of the next steps in the development of SIMPL is the inclusion of a solver to handle non-linear constraints.

Application Programming Interface.

Although SIMPL is currently a purely declarative language, it will eventually include more powerful (imperative) search constructs, such as loops and conditional statements. Meanwhile, it is possible to implement more elaborate algorithms that take advantage of SIMPL's paradigm via its Application Programming Interface (API). This API can be compiled into any customized C++ code and works similarly to other callable libraries available for commercial solvers like CPLEX or XPRESS (Dash Optimization, Inc., 2000).

Search Tree Visualization.

Once a SIMPL model finishes running, it is possible to visualize the search tree by using Leipert's VBC Tool package (Leipert). Nodes in the tree are colored red, green, black and blue to mean, respectively, pruned by infeasibility, pruned by local optimality, pruned by bound and branched on.

3.7 Conclusions and Future Work

In this paper we introduce a system for dealing with integrated models called SIMPL. The main contribution of SIMPL is to provide a user-friendly framework that generalizes many of the ways of combining Constraint Programming (CP) and Operations Research (OR) techniques when solving optimization problems. Although there exist other general-purpose systems that offer some form of hybrid modeling and solver cooperation, they do not incorporate various important features available in SIMPL.

The implementation of specialized hybrid algorithms can be a very cumbersome task. It often involves getting acquainted with the specifics of more than one type of solver (e.g. LP, CP, NLP), as well as a significant amount of computer programming, which includes coordinating the exchange of information among solvers. Clearly, a general purpose code is built at the expense of performance. Rather than defeating state-of-the-art implementations of cooperative approaches that are tailored to specific problems, SIMPL's objective is to be a generic and easy-to-use platform for the development and empirical evaluation of new ideas in the field of hybrid CP-OR algorithms.

As SIMPL is still under development, many new features and improvements to its functionality are the subject of ongoing efforts. Examples of such enhancements are: increasing the vocabulary of the language with new types of constraints; augmenting the inference capabilities of the system with the generation of cutting planes; broadening the application areas of the system by supporting other types of solvers; and providing a more powerful control over search. Finally, SIMPL is currently being used to test integrated models for a few practical optimization problems such as the lot-sizing problem of Sect. 3.5.2.

Chapter 4

An Integrated Solver for Optimization Problems

Joint work with: Ionuţ Aron and John Hooker.

Abstract: One of the central trends in the optimization community over the past several years has been the steady improvement of general-purpose solvers. A logical next step in this evolution is to combine mixed integer linear programming, global optimization, and constraint programming in a single system. Recent research in the area of integrated problem solving suggests that the right combination of different technologies can simplify modeling and speed up computation substantially. In this paper we address this goal by presenting a general purpose solver that achieves low-level integration of solution techniques with a high-level modeling language. We validate our solver with computational experiments on problems in production planning, product configuration and job scheduling. Our results indicate that an integrated approach reduces modeling effort, while solving two of the three problem classes substantially faster than state-of-the-art commercial software.

4.1 Introduction

One of the central trends in the optimization community over the past several years has been the steady improvement of general-purpose solvers. Such mixed integer solvers as CPLEX and XPRESS-MP have become significantly more effective and robust, and similar advancements have occurred in continuous global optimization (BARON, LGO) and constraint programming (CHIP, ILOG Solver). These developments promote the use of optimization and constraint solving technology, since they spare practitioners the inconvenience of acquiring and learning different software for every application.

A logical next step in this evolution is to combine mixed integer linear programming (MILP), global optimization, and constraint programming (CP) in a single system. This not only brings together a wide range of methods under one roof, but it allows users to reap the advantages of integrated problem solving. Recent research in this area suggests that the right combination of different technologies can simplify modeling and speed up computation substantially. Commercial-grade solvers are already moving in this direction, as witnessed by the ECL^{i}PS^e solver, OPL Studio, and the Mosel language, all of which combine mathematical programming and constraint programming techniques to a greater or lesser extent.

Integration is most effective when techniques interleave at a micro level. To achieve this in current systems, however, one must often write special-purpose code, which slows research and discourages application. We have attempted to address this situation by designing an architecture that achieves low-level integration of solution techniques with a high-level modeling language. The ultimate goal is

to build an integrated solver that can be used as conveniently as current mixed integer, global and constraint solvers. We have implemented our approach in a system called SIMPL, which can be read as a permuted acronym for Modeling Language for Integrated Problem Solving.

SIMPL is based on two principles: algorithmic unification and constraint-based control. *Algorithmic unification* begins with the premise that integration should occur at a fundamental and conceptual level, rather than postponed to the software design stage. Optimization methods and their hybrids should be viewed, to the extent possible, as special cases of a single solution method that can be adjusted to exploit the structure of a given problem. We find that a *search-infer-and-relax* framework serves this purpose. It encompasses a wide variety of methods, including branch-and-cut methods for integer programming, branch-and-infer methods for constraint programming, popular methods for continuous global optimization, such nogood-based methods as Benders decomposition and dynamic backtracking, and even heuristic methods such as local search and greedy randomized adaptive search procedures (GRASPs).

Constraint-based control allows the design of the model itself to tell the solver how to combine techniques so as to exploit problem structure. Highly-structured subsets of constraints are written as metaconstraints, which are similar to "global constraints" in constraint programming. Each metaconstraint invokes relaxations and inference techniques that have been devised for its particular structure. For example, a metaconstraint may be associated with a tight polyhedral relaxation from the integer programming literature and/or an effective domain filter from constraint programming. Constraints also control the search. If a branching method is used, for example, then the search branches on violated metaconstraints, and a specific branching scheme is associated with each type of constraint.

The selection of metaconstraints to formulate the problem determines how the solver combines algorithmic ideas to solve the problem. This means that SIMPL deliberately sacrifices independence of model and method: the model must be formulated with the solution method in mind. However, we believe that successful combinatorial optimization leaves no alternative. This is evident in both integer programming and constraint programming, since in either case one must carefully write the formulation to obtain tight relaxations, effective propagation, or intelligent branching. We attempt to make a virtue of necessity by explicitly providing the resources to shape the algorithm through a high-level modeling process.

We focus here on branch-and-cut, branch-and-infer, and generalized Benders methods, since these have been implemented so far in SIMPL. The system architecture is designed, however, for extension to continuous global optimization, general nogood-based methods, and heuristic methods.

After a brief survey of previous work, we summarize below the advantages of integrated problem solving and present the search-infer-and-relax framework. Following this are three examples that illustrate some modeling and algorithmic ideas that are characteristic of integrated methods. A production planning problem with semicontinuous piecewise linear costs illustrates metaconstraints and the interaction of inference and relaxation. A product configuration problem illustrates variable indices and how further inference can be derived from the solution of a relaxation. Finally, a planning and scheduling problem shows how a Benders method fits into the framework. We show how to model the three example problems in SIMPL and present computational results. We conclude with suggestions for further development. For a more detailed description of SIMPL's internal architecture, we refer the reader to Aron, Hooker, and Yunes (2004).

4.2 Previous Work

Comprehensive surveys of hybrid methods that combine CP and MILP are provided by Hooker (2000, 2002), and tutorial articles may be found Milano (2003).

Various elements of the search-infer-and-relax framework presented here were proposed by Hooker (1994, 1997, 2000, 2003), Bockmayr and Kasper (1998), Hooker and Osorio (1999), and Hooker et al. (2000). An extension to dynamic backtracking and heuristic methods is given in Hooker (2005c). A preliminary description of SIMPL appears in Aron, Hooker, and Yunes (2004). The present paper develops these ideas further and demonstrates SIMPL on a selection of problems.

Existing hybrid solvers include ECL^{i}PS^e, OPL Studio, Mosel, and SCIP. ECL^{i}PS^e is a Prolog-based constraint logic programming system that provides an interface with linear and MILP solvers (Rodošek, Wallace, and Hajian 1999; Cheadle et al. 2003; Ajili and Wallace 2003). The CP solver in ECL^{i}PS^e communicates tightened bounds to the MILP solver, while the MILP solver detects infeasibility and provides a bound on the objective function that is used by the CP solver. The optimal solution of the linear constraints in the problem can be used as a search heuristic.

OPL Studio provides an integrated modeling language that expresses both MILP and CP constraints (Van Hentenryck et al., 1999). It sends the problem to a CP or MILP solver depending on the nature of constraints. A script language allows one to write algorithms that call the CP and MILP solvers repeatedly.

Mosel is both a modeling and programming language that interfaces with various solvers, including MILP and CP solvers (Colombani and Heipcke, 2002, 2004). SCIP is a callable library that gives the user control of a solution process that can involve both CP and MILP solvers (Achterberg, 2004).

4.3 Advantages of Integrated Problem Solving

One obvious advantage of integrated problem solving is its potential for reducing computation time. Table 4.1 presents a sampling of some of the better computational results reported in the literature, divided into four groups. (a) Early efforts at integration coupled solvers rather loosely but obtained some speedup nonetheless. (b) More recent hybrid approaches combine CP or logic processing with various types of relaxations used in MILP, and they yield more substantial speedups. (c) Several investigators have used CP for column generation in a branch-and-price MILP algorithm, as proposed by Junker et al. (1999) and Yunes, Moura, and de Souza (1999). (d) Some of the largest computational improvements to date have been obtained by using generalizations of Benders decomposition to unite solution methods, as proposed by Hooker (2000). MILP is most often applied to the master problem and CP to the subproblem.

The three problems we solved with SIMPL come from Table 4.1. Our experiments confirm that an integrated approach can result in substantial speedups for two of the problems, the piecewise linear problems (Refalo, 1999) and the planning and scheduling problems (Jain and Grossmann, 2001), by several orders of magnitude in the latter case. The SIMPL modeling language allowed us to achieve these results with far less effort than the original authors, who coded special-purpose algorithms. We also solved the product configuration problems rapidly, but since recent versions of CPLEX are dramatically faster on these problems than the version used by Thorsteinsson and Ottosson (2001), we can no longer report a speedup relative to the current state of the art. Even here, however, SIMPL is more robust in terms of the node count. All of the problems in Table 4.1 can in principle be implemented in a search-infer-and-relax framework, although SIMPL in its current form is not equipped for all of them.

Aside from computational advantages, an integrated approach provides a richer modeling environment that can result in simpler models and less debugging. The full repertory of global constraints used in CP are potentially available, as well as nonlinear expressions used in continuous global optimization. Frequent use of metaconstraints not only simplifies the model but allows the solver to exploit problem structure.

Table 4.1: Sampling of computational results for integrated methods.

Source	Type of problem/method	Speedup
Loose integration of CP and MILP		
Hajian et al. (1996)	British Airways fleet assignment. CP solver provides starting feasible solution for MILP.	Twice as fast as MILP, 4 times faster than CP.
CP plus relaxations similar to those used in MILP		
Focacci, Lodi, and Milano (1999)	Lesson timetabling. Reduced-cost variable fixing using an assignment problem relaxation.	2 to 50 times faster than CP.
Refalo (1999)	Piecewise linear costs. Method similar to that described in Section 4.5.	2 to 200 times faster than MILP. Solved two instances that MILP could not solve.
Hooker and Osorio (1999)	Boat party scheduling, flow shop scheduling. Logic processing plus linear relaxation.	Solved 10-boat instance in 5 min that MILP could not solve in 12 hours. Solved flow shop instances 4 times faster than MILP.
Thorsteinsson and Ottosson (2001)	Product configuration. Method similar to that described in Section 4.6.	30 to 40 times faster than MILP (which was faster than CP).
Sellmann and Fahle (2001)	Automatic digital recording. CP plus Lagrangean relaxation.	1 to 10 times faster than MILP (which was faster than CP).
Van Hoeve (2003)	Stable set problems. CP plus semi-definite programming relaxation.	Significantly better suboptimal solutions than CP in fraction of the time.
Bollapragada, Ghattas, and Hooker (2001)	Nonlinear structural design. Logic processing plus convex relaxation.	Up to 600 times faster than MILP. Solved 2 problems in < 6 min that MILP could not solve in 20 hours.
Beck and Refalo (2003)	Scheduling with earliness & tardiness costs.	Solved 67 of 90 instances, while CP solved only 12.
CP-based branch and price		
Easton, Nemhauser, and Trick (2002)	Traveling tournament scheduling.	First to solve 8-team instance.
Yunes, Moura, and de Souza (2005)	Urban transit crew management.	Solved problems with 210 trips, while traditional branch and price could accommodate only 120 trips.
Benders-based integration of CP and MILP		
Jain and Grossmann (2001)	Min-cost planning and disjunctive scheduling. MILP master problem, CP subproblem (Section 4.7).	20 to 1000 times faster than CP, MILP.
Thorsteinsson (2001)	Jain & Grossmann problems. Branch and check.	Additional factor of 10 over Jain & Grossmann (2001).
Timpe (2002)	Polypropylene batch scheduling at BASF. MILP master, CP subproblem.	Solved previously insoluble problem in 10 min.
Benoist, Gaudin, and Rottembourg (2002)	Call center scheduling. CP master, LP subproblem.	Solved twice as many instances as traditional Benders.
Hooker (2004)	Min-cost and min-makespan planning and cumulative scheduling. MILP master, CP subproblem.	100 to 1000 times faster than CP, MILP. Solved significantly larger instances.
Hooker (2005a)	Min-tardiness planning & cumulative scheduling. MILP master, CP subproblem.	10 to >1000 times faster than CP, MILP when minimizing # late jobs; ~ 10 times faster when minimizing total tardiness, much better solutions when suboptimal.
Rasmussen and Trick (2005)	Sports scheduling to minimize # of consecutive home or away games.	Speedup of several orders of magnitude compared to previous state of the art.

This implies a different style of modeling than is customary in mathematical programming, which writes all constraints using a few primitive terms (equations, inequalities, and some algebraic expressions). Effective integrated modeling draws from a large library of metaconstraints and presupposes that the user has some familiarity with this library. For instance, the library may contain a constraint that defines piecewise linear costs, a constraint that requires flow balance in a network, a constraint that prevents scheduled jobs from overlapping, and so forth. Each constraint is written with parameters that specify the shape of the function, the structure of the network, or the processing times of the jobs. When sitting down to formulate a model, the user would browse the library for constraints that appear to relate to the problem at hand.

Integrated modeling therefore places on the user the burden of identifying problem structure, but in so doing it takes full advantage of the human capacity for pattern recognition. Users identify highly structured subsets of constraints, which allows the solver to apply the best known analysis of these structures to solve the problem efficiently. In addition, only certain metaconstraints tend to occur in a given problem domain. This means that only a relevant portion of the library must be presented to a practitioner in that domain.

4.4 The Basic Algorithm

The search-infer-and-relax algorithm can be summarized as follows:

Search. The search proceeds by solving problem *restrictions*, each of which is obtained by adding constraints to the problem. The motivation is that restrictions may be easier to solve than the original problem. For example, a branch-and-bound method for MILP enumerates restrictions that correspond to nodes of the search tree. Branch-and-infer methods in CP do the same. Benders decomposition enumerates restrictions in the form of subproblems (slave problems). If the search is exhaustive, the best feasible solution of a restriction is optimal in the original problem. A search is exhaustive when the restrictions have feasible sets whose union is the feasible set of the original problem.

Infer. Very often the search can be accelerated by inferring valid constraints from the current problem restriction, which are added to the constraint set. MILP methods infer constraints in the form of cutting planes. CP methods infer smaller variable domains from individual constraints. (A variable's domain is the set of values it can take.) Benders methods infer valid cuts from the subproblem by solving its dual.

One can often exploit problem structure by designing specialized inference methods for certain metaconstraints or highly structured subsets of constraints. Thus MILP generates specialized cutting planes for certain types of inequality sets (e.g. knapsacks). CP applies specialized domain reduction or "filtering" algorithms to such commonly used global constraints as *all-different*, *element* and *cumulative*. Benders cuts commonly exploit the structure of the subproblem.

Inference methods that are applied only to subsets of constraints often miss implications of the entire constraint set, but this can be partially remedied by *constraint propagation*, a fundamental technique of CP. For example, domains that are reduced by a filtering algorithm for one constraint can serve as the starting point for the domain reduction algorithm applied to the next constraint, and so forth. Thus the results of processing one constraint are "propagated" to the next constraint.

Relax. It is often useful to solve a relaxation of the current problem restriction, particularly when the restriction is too hard to solve. The relaxation can provide a bound on the optimal value, perhaps

a solution that happens to be feasible in the original problem, and guidance for generating the next problem restriction.

In MILP, one typically solves linear programming or Lagrangean relaxations to obtain bounds or solutions that may happen to be integer. The solution of the relaxation also helps to direct the search, as for instance when one branches on a fractional variable. In Benders decomposition, the master problem is the relaxation. Its solution provides a bound on the optimum and determines the next restriction (subproblem) to be solved.

Like inference, relaxation is very useful for exploiting problem structure. For example, if the model identifies a highly structured subset of inequality constraints (by treating them as a single metaconstraint), the solver can generate a linear relaxation for them that contains specialized cutting planes. This allows one to exploit structure that is missed even by current MILP solvers. In addition, such global constraints as *all-different*, *element* and *cumulative* can be given specialized linear relaxations.

4.5 Example: Piecewise Linear Functions

A simple production planning example with piecewise linear functions illustrates integrated modeling as well as the search-infer-and-relax process. The approach taken here is similar to that of Refalo (1999) Ottosson, Thorsteinsson, and Hooker (1999, 2002).

The objective is to manufacture several products at a plant of limited capacity C so as to maximize net income. Each product must be manufactured in one of several production modes (small scale, medium scale, large scale, etc.), and only a certain range of production quantities are possible for each mode. Thus if x_i units of product i are manufactured in mode k, $x_i \in [L_{ik}, U_{ik}]$. The net income $f_i(x_i)$ derived from making product i is linear in each interval $[L_{ik}, U_{ik}]$, which means that f_i is a semicontinuous piecewise linear function (Fig. 4.1):

$$f_i(x_i) = \frac{U_{ik} - x_i}{U_{ik} - L_{ik}} c_{ik} + \frac{x_i - L_{ik}}{U_{ik} - L_{ik}} d_{ik}, \quad \text{if } x_i \in [L_{ik}, U_{ik}] \tag{4.1}$$

Making none of product i corresponds to mode $k = 0$, for which $[L_{i0}, U_{i0}] = [0, 0]$.

An integer programming model for this problem introduces 0-1 variables y_{ik} to indicate the production mode of each product. The functions f_i are modeled by assigning weights λ_{ij}, μ_{ik} to the endpoints of each interval k. The model is

$$\begin{aligned}
&\max \sum_{ik} \lambda_{ik} c_{ik} + \mu_{ik} d_{ik} \\
&\sum_i x_i \le C \\
&x_i = \sum_k \lambda_{ik} L_{ik} + \mu_{ik} U_{ik}, \text{ all } i \\
&\sum_k \lambda_{ik} + \mu_{ik} = 1, \text{ all } i \\
&0 \le \lambda_{ik} \le y_{ik}, \text{ all } i, k \\
&0 \le \mu_{ik} \le y_{ik}, \text{ all } i, k \\
&\sum_k y_{ik} = 1, \text{ all } i \\
&y_{ik} \in \{0, 1\}, \text{ all } i, k
\end{aligned} \tag{4.2}$$

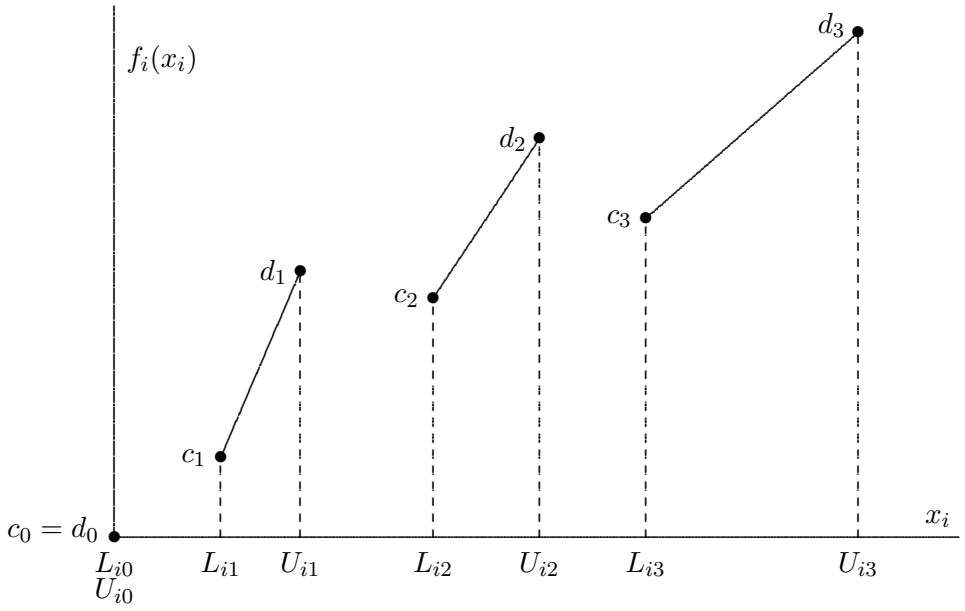

Figure 4.1: A semicontinuous piecewise linear function $f_i(x_i)$.

If desired one can identify $\lambda_{i0}, \mu_{i0}, \lambda_{i1}, \mu_{i1}, \ldots$ as a specially ordered set of type 2 for each product i. However, specially ordered sets are not particularly relevant here, since the adjacent pair of variables $\mu_{ik}, \lambda_{i,k+1}$ are never both positive for any k.

The most direct way to write an integrated model for this problem is to use conditional constraints of the form $A \Rightarrow B$, which means that if the antecedent A is true, then the consequent B is enforced. The piecewise linear cost function $f_i(x_i)$ can be coded much as it appears in (4.1):

$$\begin{aligned}
&\max \sum_i u_i \\
&\sum_i x_i \le C && (a) \\
&(x_i \in [L_{ik}, U_{ik}]) \Rightarrow \left(u_i = \frac{U_{ik} - x_i}{U_{ik} - L_{ik}} c_{ik} + \frac{x_i - L_{ik}}{U_{ik} - L_{ik}} d_{ik} \right), \text{ all } i, k && (b) \\
&x_i \in \bigcup_k [L_{ik}, U_{ik}], \text{ all } i && (c)
\end{aligned} \tag{4.3}$$

Note that no discrete variables are required, and the model is quite simple.

The problem can be solved by branching on the continuous variables x_i in a branch-and-bound method. In this case the domain of x_i is an interval of real numbers. The search branches on an x_i by splitting its domain into two or more smaller intervals, much as is done in continuous global solvers, so that the domains become smaller as one descends into the search tree. The solver processes the conditional constraints (b) by adding the consequent to the constraint set whenever the current domain of x_i lies totally within $[L_{ik}, U_{ik}]$.

Although model (4.3) is a perfectly legitimate approach to solving the problem, it does not fully exploit the problem's structure. For each product i, the point (x_i, u_i) must lie on one of the line segments defined by consequents of (b). If the solver were aware of this fact, it could construct a tight linear relaxation by requiring (x_i, u_i) to lie in the convex hull of these line segments, thus resulting in substantially faster solution.

This is accomplished by equipping the modeling language with metaconstraint *piecewise* to model continuous or semicontinuous piecewise linear functions. A single piecewise constraint represents the constraints in (b) that correspond to a given i. The model (4.3) becomes

$$\begin{aligned} &\max \sum_i u_i \\ &\sum_i x_i \le C \qquad (a) \\ &\text{piecewise}(x_i, u_i, L_i, U_i, c_i, d_i), \text{ all } i \quad (b) \end{aligned} \tag{4.4}$$

Here L_i is an array containing $L_{i0}, L_{i1}, \ldots$, and similarly for U_i, c_i, and d_i. Each piecewise constraint enforces $u_i = f_i(x_i)$.

In general the solver has a library of metaconstraints that are appropriate to common modeling situations. Typically some constraints are written individually, as is (a) above, while others are collected under one or more metaconstraints in order simplify the model and allow the solver to exploit problem structure. It does so by applying inference methods to each metaconstraint, relaxing it, and branching in an intelligent way when it is not satisfied. In each case the solver exploits the peculiar structure of the constraint.

Let us suppose the solver is instructed to solve the production planning problem by branch and bound, which defines the *search* component of the algorithm. The search proceeds by enumerating restrictions of the problem, each one corresponding to a node of the search tree. At each node, the solver *infers* a domain $[a_i, b_i]$ for each variable x_i. Finally, the solver generates bounds for the branch-and-bound mechanism by solving a *relaxation* of the problem. It branches whenever a constraint is violated by the solution of the relaxation, and the nature of the branching is dictated by the constraint that is violated.

It is useful to examine these steps in more detail. At a given node of the search tree, the solver first applies inference methods to each constraint. Constraint (a) triggers a simple form of *interval propagation*. The upper bound b_i of each x_i's domain is adjusted to become $\min\left\{b_i, C - \sum_{j \ne i} a_j\right\}$. Constraint (b) can also reduce the domain of x_i, as will be seen shortly. Domains reduced by one constraint can be cycled back through the other constraint for possible further reduction. As branching and propagation reduce the domains, the problem relaxation becomes progressively tighter until its solution is feasible in the original problem.

The solver creates a relaxation at each node of the search tree by pooling relaxations of the various constraints. It relaxes each constraint in (b) by generating linear inequalities to describe the convex hull of the graph of each f_i, as illustrated in Fig. 4.2. The fact that x_i is restricted to $[a_i, b_i]$ permits a tighter relaxation, as shown in the figure. Similar reasoning reduces the domain $[a_i, b_i]$ of x_i to $[L_{i1}, b_i]$. The linear constraint (a) also generates a linear relaxation, namely itself. These relaxations, along with the domains, combine to form a linear relaxation of the entire problem:

$$\begin{aligned} &\max \sum_i u_i \\ &\sum_i x_i \le C \\ &\text{conv}(\text{piecewise}(x_i, u_i, L_i, U_i, c_i, d_i)), \text{ all } i \\ &a_i \le x_i \le b_i, \text{ all } i \end{aligned} \tag{4.5}$$

where *conv* denotes the convex hull description just mentioned.

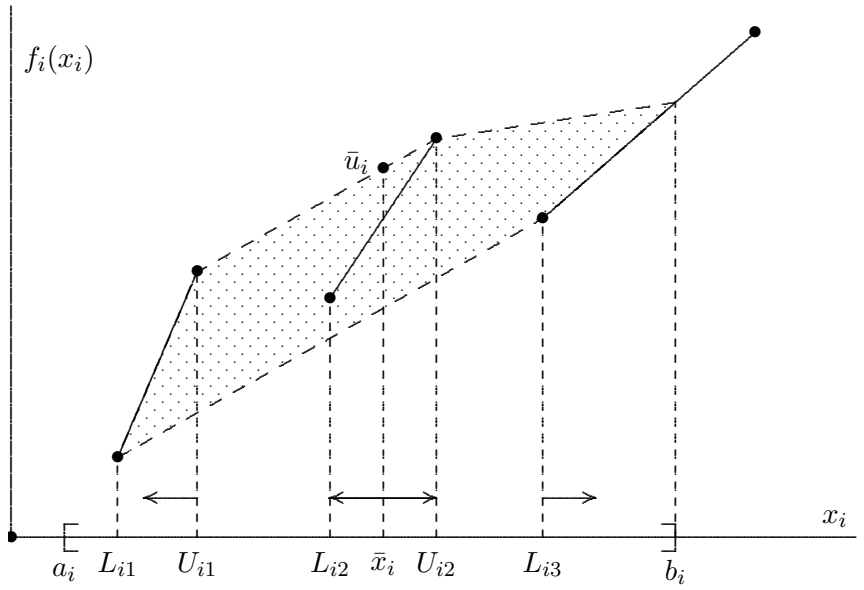

Figure 4.2: Convex hull relaxation (shaded area) of $f_i(x_i)$ when x_i has domain $[a_i, b_i]$.

The solver next finds an optimal solution $(\overline{x}_i, \overline{u}_i)$ of the relaxation (4.5) by calling a linear programming plug-in. This solution will necessarily satisfy (a), but it may violate (b) for some product i, for instance if $\overline{x}_i$ is not a permissible value of x_i, or $\overline{u}_i$ is not the correct value of $f_i(\overline{x}_i)$. The latter case is illustrated in Fig. 4.2, where the search creates three branches by splitting the domain of x_i into three parts: $[L_{i2}, U_{i2}]$, everything below U_{i1}, and everything above L_{i3}. Note that in this instance the linear relaxation at all three branches will be exact, so that no further branching will be necessary.

The problem is therefore solved by combining ideas from three technologies: search by splitting intervals, from continuous global optimization; domain reduction, from constraint programming; and polyhedral relaxation, from integer programming.

4.6 Example: Variable Indices

A variable index is a versatile modeling device that is readily accommodated by a search-infer-and-relax solver. If an expression has the form u_y, where y is a variable, then y is a *variable index* or variable subscript. A simple product configuration problem (Thorsteinsson and Ottosson, 2001) illustrates how variable indices can be used in a model and processed by a solver.

The problem is to choose an optimal configuration of components for a product, such as a personal computer. For each component i, perhaps a memory chip or power supply, one must decide how many q_i to install and what type t_i to install. Only one type of each component may be used. The types correspond to different technical specifications, and each type k of component i supplies a certain amount a_{ijk} of attribute j. For instance, a given type of memory chip might supply a certain amount of memory, generate a certain amount of heat, and consume a certain amount of power; in the last case, $a_{ijk} < 0$ to represent a negative supply. There are lower and upper bounds L_j, U_j on each attribute j. Thus there may be a lower bound on total memory, an upper bound on heat generation, a lower bound of zero on net power supply, and so forth. Each unit of attribute j produced incurs a (possibly negative) penalty c_j.

A straightforward integer programming model introduces 0-1 variables x_{ik} to indicate when type k of component i is chosen. The total penalty is $\sum_j c_j v_j$, where v_j is the amount of attribute j

produced. The quantity v_j is equal to $\sum_{ik} a_{ijk} q_i x_{ik}$. Since this is a nonlinear expression, the variables q_i are disaggregated, so that q_{ik} becomes the number of units of type k of component i. The quantity v_j is now given by the linear expression $\sum_{ik} a_{ijk} q_{ik}$. A big-M constraint can be used to force q_{ij} to zero when $x_{ij} = 0$. The model becomes,

$$\begin{aligned} &\min \sum_j c_j v_j \\ &v_j = \sum_{ik} a_{ijk} q_{ik}, \text{ all } j \quad (a) \\ &L_j \le v_j \le U_j, \text{ all } j \quad (b) \\ &q_{ik} \le M_i x_{ik}, \text{ all } i,k \quad (c) \\ &\sum_k x_{ik} = 1, \text{ all } i \quad (d) \end{aligned} \tag{4.6}$$

where each x_{ij} is a 0-1 variable, each q_{ij} is integer, and M_i is an upper bound on q_i. If the MILP modeling language accommodates specially ordered sets of nonbinary variables, the variables x_{ij} can be eliminated and constraints (c) and (d) replaced by a stipulation that $\{q_{i1}, q_{i2}, \ldots\}$ is a specially ordered set of type 1 for each i.

An integrated model uses the original notation t_i for the type of component i, without the need for 0-1 variables or disaggregation. The key is to permit t_i to appear as a subscript:

$$\begin{aligned} &\min \sum_j c_j v_j \\ &v_j = \sum_i q_i a_{ijt_i}, \text{ all } j \end{aligned} \tag{4.7}$$

where the bounds L_j, U_j are reflected in the initial domain assigned to v_j.

The modeling system automatically decodes variably indexed expressions with the help of the *element* constraint, which is frequently used in CP modeling. In this case the variably indexed expressions occur in *indexed linear* expressions of the form

$$\sum_i q_i a_{ijt_i} \tag{4.8}$$

where each q_i is an integer variable and each t_i a discrete variable. Each term $q_i a_{ijt_i}$ is automatically replaced with a new variable z_{ij} and the constraint

$$\text{element}(t_i, (q_i a_{ij1}, \ldots, q_i a_{ijn}), z_{ij}) \tag{4.9}$$

This constraint in effect forces $z_{ij} = q_i a_{ijt_i}$. The solver can now apply a domain reduction or "filtering" algorithm to (4.9) and generate a relaxation for it.

For a given j, filtering for (4.9) is straightforward. If z_{ij}'s domain is $[\underline{z}_{ij}, \overline{z}_{ij}]$, t_i's domain is D_{t_i}, and q_i's domain is $\{\underline{q}_i, \underline{q}_i + 1, \ldots, \overline{q}_i\}$ at any point in the search, then the reduced domains $[\underline{z}'_{ij}, \overline{z}'_{ij}]$, D'_{t_i}, and $\{\underline{q}'_i, \ldots, \overline{q}'_i\}$ are given by

$$\begin{aligned} &\underline{z}'_{ij} = \min\left\{\underline{z}_{ij}, \min_k\left\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\right\}\right\}, \quad \overline{z}'_{ij} = \max\left\{\overline{z}_{ij}, \max_k\left\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\right\}\right\}, \\ &D'_{t_i} = D_{t_i} \cap \left\{k \;\middle|\; [\underline{z}'_{ij}, \overline{z}'_{ij}] \cap \left[\min\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\}, \max\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\}\right] \neq \emptyset\right\} \\ &\underline{q}'_i = \min\left\{\underline{q}_i, \min_k\left\{\left\lceil\frac{\underline{z}'_{ij}}{a_{ijk}}\right\rceil, \left\lceil\frac{\overline{z}'_{ij}}{a_{ijk}}\right\rceil\right\}\right\}, \quad \overline{q}'_i = \max\left\{\overline{q}_i, \max_k\left\{\left\lfloor\frac{\underline{z}'_{ij}}{a_{ijk}}\right\rfloor, \left\lfloor\frac{\overline{z}'_{ij}}{a_{ijk}}\right\rfloor\right\}\right\} \end{aligned}$$

Since (4.9) implies a disjunction $\bigvee_{k\in D_{t_i}}(z_{ij} = a_{ijk}q_i)$, it can be given the standard convex hull relaxation for a disjunction which in this case simplifies to

$$z_{ij} = \sum_{k\in D_{t_i}} a_{ijk}q_{ik}, \quad q_i = \sum_{k\in D_{t_i}} q_{ik}$$

where $q_{ik} \geq 0$ are new variables.

If there is a lower bound L on the expression (4.8), the relaxation used by Thorsteinsson and Ottosson (2001) can be strengthened with integer knapsack cuts (and similarly if there is an upper bound). Since

$$\sum_i q_i a_{ijt_i} = \sum_i \sum_{k\in D_{t_i}} a_{ijk}q_{ik} \leq \sum_i \max_{k\in D_{t_i}}\{a_{ijk}\} \sum_{k\in D_{t_i}} q_{ik} = \sum_i \max_{k\in D_{t_i}}\{a_{ijk}\}q_i$$

the lower bound L on (4.8) yields the valid inequality

$$\sum_i \max_{k\in D_{t_i}}\{a_{ijk}\}q_i \geq L \tag{4.10}$$

Since the q_is are general integer variables, integer knapsack cuts can be generated for (4.10).

Based on these ideas, the automatically generated relaxation of (4.7) becomes

$$\begin{aligned}
&\min \sum_j c_j v_j \\
&v_j = \sum_i \sum_{k\in D_{t_i}} a_{ijk}q_{ik}, \quad \text{all } j \\
&q_i = \sum_{k\in D_{t_i}} q_{ik}, \quad \text{all } i \\
&L_j \leq v_j \leq U_j, \quad \text{all } j \\
&\underline{q}_i \leq q_i \leq \overline{q}_i, \quad \text{all } i \\
&\text{knapsack cuts for } \sum_i \max_{k\in D_{t_i}}\{a_{ijk}\}q_i \geq L_j, \text{ all } j \\
&\text{knapsack cuts for } \sum_i \min_{k\in D_{t_i}}\{a_{ijk}\}q_i \leq U_j, \text{ all } j \\
&q_{ik} \geq 0, \quad \text{all } i,k
\end{aligned} \tag{4.11}$$

There is also an opportunity for *post-relaxation inference*, which in this case takes the form of reduced cost variable fixing. Suppose the best feasible solution found so far has value z^*, and let $\hat{z}$ be the optimal value of (4.11). If $\hat{z} + r_{ik} \geq z^*$, where r_{ik} is the reduced cost of q_{ik} in the solution of (4.11), then k can be removed from the domain of t_i. In addition, if $r_{ik} > 0$, one can infer

$$\overline{q}_i \leq \max_{k\in D_{t_i}} \left\{ \left\lfloor \frac{z^* - \hat{z}}{r_{ik}} \right\rfloor \right\}, \quad \text{all } i$$

Post-relaxation inference can take other forms as well, such as the generation of separating cuts.

The problem can be solved by branch and bound. In this case, we can start by branching on the domain constraints $t_i \in D_{t_i}$. Since t_i does not appear in the linear relaxation, it does not have a determinate value until it is fixed by branching. The domain constraint $t_i \in D_{t_i}$ is viewed as unsatisfied as long as t_i is undetermined. The search branches on $t_i \in D_{t_i}$ by splitting D_{t_i} into two subsets. Branching continues until all the D_{t_i} are singletons, or until at most one q_{ik} (for $k \in D_{t_i}$) is positive for each i. At that point we check if all q_i variables are integer and branch on q if necessary.

4.7 Example: Logic-based Benders Decomposition

Nogood-based methods search the solution space by generating a *nogood* each time a candidate solution is examined. The nogood is a constraint that excludes the solution just examined, and perhaps other solutions that can be no better. The next solution enumerated must satisfy the nogoods generated so far. The search is exhaustive when the nogood set becomes infeasible.

Benders decomposition is a special type of nogood-based method in which the nogoods are Benders cuts and the master problem contains all the nogoods generated so far. In classical Benders, the subproblem (slave problem) is a linear or nonlinear programming problem, and Benders cuts are obtained by solving its dual—or in the nonlinear case by deriving Lagrange multipliers. *Logic-based* Benders generalizes this idea to an arbitrary subproblem by solving the *inference dual* of the subproblem (Hooker and Yan, 1995; Hooker, 1996, 1999).

A simple planning and scheduling problem illustrates the basic idea (Hooker, 2000; Jain and Grossmann, 2001). A set of n jobs must be assigned to machines, and the jobs assigned to each machine must be scheduled subject to time windows. Job j has release time r_j, deadline d_j, and processing time p_{ij} on machine i. It costs c_{ij} to process job j on machine i. It generally costs more to run a job on a faster machine. The objective is to minimize processing cost.

We now describe the MILP model used by Jain and Grossmann (2001). Let the binary variable x_{ij} be one if job j is assigned to machine i, and let the binary variable $y_{jj'}$ be one if both j and j' are assigned to the same machine and j finishes before j' starts. In addition, let t_j be the time at which job j starts. The MILP model is as follows.

$$
\begin{aligned}
&\min \sum_{ij} c_{ij} x_{ij} \\
&r_j \le t_j \le d_j - \sum_i p_{ij} x_{ij}, \text{ all } j && \text{(a)} \\
&\sum_i x_{ij} = 1, \text{ all } j && \text{(b)} \\
&y_{jj'} + y_{j'j} \le 1, \text{ all } j' > j && \text{(c)} \\
&y_{jj'} + y_{j'j} + x_{ij} + x_{i'j'} \le 2, \text{ all } j' > j,\ i' \ne i && \text{(d)} \\
&y_{jj'} + y_{j'j} \ge x_{ij} + x_{ij'} - 1, \text{ all } j' > j,\ i && \text{(e)} \\
&t_{j'} \ge t_j + \sum_i p_{ij} x_{ij} - \mathcal{U}(1 - y_{jj'}), \text{ all } j' \ne j && \text{(f)} \\
&\sum_j p_{ij} x_{ij} \le \max_j\{d_j\} - \min_j\{r_j\}, \text{ all } i && \text{(g)} \\
&x_{ij} \in \{0,1\},\ \ y_{jj'} \in \{0,1\} \text{ for all } j' \ne j
\end{aligned}
\tag{4.12}
$$

Constraint (a) defines lower and upper bounds on the start time of job j, and (b) makes sure every job is assigned to some machine. Constraints (c) and (d) are logical cuts which significantly reduce solution time. Constraint (e) defines a logical relationship between the assignment (x) and sequencing (y) variables, and (f) ensures start times are consistent with the value of the sequencing variables y ($\mathcal{U} = \sum_j \max_i\{p_{ij}\}$). Finally, (g) are valid cuts that tighten the linear relaxation of the problem. There is also a continuous-time MILP model suggested by Türkay and Grossmann (1996), but computational testing indicates that it is much harder to solve than (4.12) (Hooker, 2004).

A hybrid model can be written with the *cumulative* metaconstraint, which is widely used in constraint programming for "cumulative" scheduling, in which several jobs can run simultaneously but

subject to a resource constraint and time windows. Let t_j be the time at which job j starts processing and u_{ij} the rate at which job j consumes resources when it is running on machine i. The constraint

$$\text{cumulative}(t, p_i, u_i, U_i)$$

requires that the total rate at which resources are consumed on machine i be always less than or equal to U_i. Here $t = (t_1, \ldots, t_n)$, $p_i = (p_{i1}, \ldots, p_{in})$, and similarly for u_i.

In the present instance, jobs must run sequentially on each machine. Thus each job j consumes resources at the rate $u_{ij} = 1$, and the resource limit is $U_i = 1$. Thus if y_j is the machine assigned to job j, the problem can be written

$$\begin{aligned} &\min \sum_j c_{y_j j} \\ &r_j \le t_j \le d_j - p_{y_j j}, \text{ all } j \\ &\text{cumulative}((t_j | y_j = i), (p_{ij} | y_j = i), e, 1), \text{ all } i \end{aligned} \tag{4.13}$$

where e is a vector of ones.

This model is adequate for small problems, but solution can be dramatically accelerated by decomposing the problem into an assignment portion to be solved by MILP and a subproblem to be solved by CP. The assignment portion becomes the Benders master problem, which allocates job j to machine i when $x_{ij} = 1$:

$$\begin{aligned} &\min \sum_{ij} c_{ij} x_{ij} \\ &\sum_i x_{ij} = 1, \text{ all } j \\ &\text{relaxation of subproblem} \\ &\text{Benders cuts} \\ &x_{ij} \in \{0, 1\} \end{aligned} \tag{4.14}$$

The solution $\overline{x}$ of the master problem determines the assignment of jobs to machines. Once these assignments are made, the problem (4.13) separates into a scheduling feasibility problem on each machine i:

$$\begin{aligned} &r_j \le t_j \le d_j - p_{\overline{y}_j j}, \text{ all } j \\ &\text{cumulative}((t_j | \overline{y}_j = i), (p_{ij} | \overline{y}_j = i), e, 1) \end{aligned} \tag{4.15}$$

where $\overline{y}_j = i$ when $\overline{x}_{ij} = 1$. If there is a feasible schedule for every machine, the problem is solved. If, however, the scheduling subproblem (4.15) is infeasible on some machine i, a Benders cut is generated to rule out the solution $\overline{x}$, perhaps along with other solutions that are known to be infeasible. The Benders cuts are added to the master problem, which is re-solved to obtain another assignment $\overline{x}$.

The simplest sort of Benders cut for machine i rules out assigning the same set of jobs to that machine again:

$$\sum_{j \in J_i} (1 - x_{ij}) \ge 1 \tag{4.16}$$

where $J_i = \{j \mid \overline{x}_{ij} = 1\}$. A stronger cut can be obtained, however, by deriving a smaller set J_i of jobs that are actually responsible for the infeasibility. This can be done by removing elements from J_i one at a time, and re-solving the subproblem, until the scheduling problem becomes feasible (Hooker, 2005c). Another approach is to examine the proof of infeasibility in the subproblem and note which jobs actually play a role in the proof (Hooker, 2005a). In CP, an infeasibility proof generally takes the

form of edge finding techniques for domain reduction, perhaps along with branching. Such a proof of infeasibility can be regarded as a solution of the subproblem's *inference dual*. (In linear programming, the inference dual is the classical linear programming dual.) Logic-based Benders cuts can also be developed for planning and scheduling problems in which the subproblem is an optimization rather than a feasibility problem. This occurs, for instance, in minimum makespan and minimum tardiness problems (Hooker, 2004).

It is computationally useful to strengthen the master problem with a relaxation of the subproblem. The simplest relaxation requires that the processing times of jobs assigned to machine i fit between the earliest release time and latest deadline:

$$\sum_j p_{ij} x_{ij} \le \max_j\{d_j\} - \min_j\{r_j\} \tag{4.17}$$

A Benders method (as well as any nogood-based method) fits easily into the search-infer-and-relax framework. It solves a series of problem restrictions in the form of subproblems. The search is directed by the solution of a relaxation, which in this case is the master problem. The inference stage generates Benders cuts.

The decomposition is communicated to the solver by writing the model

$$\begin{aligned}
&\min \sum_{ij} c_{ij} x_{ij} && \text{(a)}\\
&\sum_i x_{ij} = 1, \text{ all } j && \text{(b)}\\
&(x_{ij} = 1) \Leftrightarrow (y_j = i)\,, \text{ all } i, j && \text{(c)}\\
&r_j \le t_j \le d_j - p_{y_j j}, \text{ all } j && \text{(d)}\\
&\text{cumulative}((t_j | y_j = i), (p_{ij} | \overline{y}_j = i), e, 1), \text{ all } i && \text{(e)}
\end{aligned} \tag{4.18}$$

where the domain of each x_{ij} is $\{0, 1\}$. Each constraint is associated with a relaxation parameter and an inference parameter. The relaxation parameters for the constraints (b) and (d) will indicate that these constraints contribute to the MILP master relaxation of the problem. Note that (d) and (e) are part of the Benders subproblem. The relaxation parameter for (e) will add the inequalities (4.17) to the linear relaxation. The inference parameter for (e) will specify the type of Benders cuts to be generated. When the solver is instructed to use a Benders method, it automatically adds the Benders cuts to the relaxation. For more details on how these parameters are stated, see Section 4.8.3.

4.8 Computational Experiments

In the next three subsections, we implement each of the three examples described in Sections 4.5 through 4.7, respectively, and compare the performance of the two optimization models proposed in each case: a traditional MILP model (as implemented by CPLEX 9.0), and an integrated model (as implemented by SIMPL).

It is important to note that SIMPL is still a research code and has not been optimized for speed. We report both the number of search nodes and the computation time, but the node count may be a more important indication of performance at the present stage of development. The amount of time SIMPL spends per node can be reduced by optimizing the code, whereas the node count is more a function of the underlying algorithm. In any event SIMPL requires substantially less time, as well as fewer nodes, on two of the three problems classes studied here. Its performance also appears to be more robust, with respect to the number of nodes, on all three problems.

We ran all the experiments reported in this section on a Pentium 4, 3.7 GHz with 4GB of RAM, running Gentoo Linux kernel 2.6.12.5. We used CPLEX 9.0 as the LP solver and ECL^{i}PS^e 5.8 as the CP solver in SIMPL. For simplicity, when showing the SIMPL code of each model we omit the data and variable declaration statements.

4.8.1 Production Planning

The SIMPL code that corresponds to the integrated model (4.4) of the production planning problem is shown below.

```
01. OBJECTIVE
02.   maximize sum i of u[i]
03. CONSTRAINTS
04.   capacity means {
05.     sum i of x[i] <= C
06.     relaxation = { lp, cp } }
07.   piecewisectr means {
08.     piecewise(x[i],u[i],L[i],U[i],c[i],d[i]) forall i
09.     relaxation = { lp, cp } }
10. SEARCH
11.   type = { bb:bestdive }
12.   branching = { piecewisectr:most }
```

Lines 06 and 09 of the above code tell SIMPL that those constraints should be posted to both the linear programming (`lp`) and constraint programming (`cp`) relaxations/solvers. Recall that the linear programming relaxation of the i^{th} piecewise constraint is the collection of inequalities on x_i and u_i that define the convex hull of their feasible values in the current state of the search. Line 11 indicates that we use branch-and-bound (`bb`), select the current active node with the best lower bound, and dive from it until we reach a leaf node (keyword `bestdive`). Finally, in line 12 we say that the branching strategy is to branch on the piecewise constraint with the largest degree of violation (keyword `most`). The amount of violation is calculated by measuring how far the LP relaxation values of x_i and u_i are from the closest linear piece of the function.

We ran both the pure MILP model (4.2) and the above integrated model over 10 randomly generated instances with the number of products n ranging from 5 to 50. In all instances, products have the same cost structure with five production modes. For the purpose of symmetry breaking, the two models also include constraints of the form $x_i \leq x_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. The number of search nodes and CPU time (in seconds) required to solve each of the ten instances to optimality are shown in Table 4.2. As the number of products increases, one can see that the number of search nodes required by a pure MILP approach can be more than 500 times larger than the number of nodes required by the integrated approach. As a result, the difference in computing time is also significant.

4.8.2 Product Configuration

The product configuration problem of Section 4.6 can be coded in SIMPL as follows.

```
01. OBJECTIVE
02.   minimize sum j of c[j]*v[j]
03. CONSTRAINTS
04.   usage means {
```

Table 4.2: Production planning: search nodes and CPU time.

Number of	CPLEX		SIMPL	
Products	Nodes	Time (s)	Nodes	Time (s)
5	93	0.03	13	0.09
10	423	0.12	40	0.35
15	1321	0.34	33	0.58
20	4573	1.14	53	1.00
25	5105	2.43	19	0.30
30	4504	2.10	43	0.82
35	6089	3.30	33	1.11
40	7973	4.06	36	1.15
45	23414	14.72	40	1.76
50	18795	9.45	47	1.77

```
05.     v[j] = sum i of q[i]*a[i][j][t[i]] forall j
06.     relaxation = { lp, cp }
07.     inference = { knapsack } }
08.   quantities means {
09.     q[1] >= 1 => q[2] = 0
10.     relaxation = { lp, cp } }
11.   types means {
12.     t[1] = 1 => t[2] in {1,2}
13.     t[3] = 1 => (t[4] in {1,3} and t[5] in {1,3,4,6} and t[6] = 3)
14.     relaxation = { lp, cp } }
15. SEARCH
16.   type = { bb:bestdive }
17.   branching = { quantities, t:most, q:least:triple, types:most }
18.   inference = { q:redcost }
```

For our computational experiments, we used ten of the problem instances proposed by Thorsteinsson and Ottosson (2001), which have 26 components, up to 30 types per component, and 8 attributes. These instances have a few extra logical constraints on the q_i and t_i variables, which are implemented in lines 08 through 14 above. These constraints are also added to the MILP model (4.6).

For the `usage` constraints, note the variable subscript notation in line 05 and the statement that tells it to infer integer knapsack cuts (as described in Section 4.6) in line 07 (this would be the default behavior anyway). We define our branching strategy in line 17 as follows: we first try to branch on the q-implication (`quantities`), then on the indomain constraints of the t variables (most violated first), followed by the indomain constraints on the q variables (least violated first), and finally on the implications on the t variables (`types`, most violated first). The indomain constraint of a t_i variable is violated if its domain is not a singleton and two or more of the corresponding q_{ik} variables have a positive value in the LP relaxation (see Section 4.6 for the relationship between t_i and q_{ik}, and note that this relationship is automatically created by SIMPL without any extra burden to the user). The keyword `triple` that appears after `q:least` in line 17 indicates that we branch on q_i as suggested by Thorsteinsson and Ottosson (2001): let $\overline{q}_i$ be the closest integer to the fractional value of q_i in the current solution of the LP relaxation; we create up to three descendents of the current node by adding

Table 4.3: Product configuration: search nodes and CPU time.

	CPLEX		SIMPL	
Instance	Nodes	Time (s)	Nodes	Time (s)
1	1	0.06	61	6.64
2	1	0.08	34	3.08
3	184	0.79	164	20.42
4	1	0.04	27	2.53
5	723	4.21	30	2.91
6	1	0.05	30	1.99
7	111	0.59	32	2.97
8	20	0.19	29	2.94
9	20	0.17	18	0.97
10	1	0.03	32	2.60

each of the following constraints in turn (if possible): $q_i = \overline{q}_i$, $q_i \leq \overline{q}_i - 1$ and $q_i \geq \overline{q}_i + 1$. Finally, the post-relaxation inference using reduced costs is turned on for the q variables in line 18.

The number of search nodes and CPU time (in seconds) required to solve each of the ten instances to optimality are shown in Table 4.3. Although the integrated model solves considerably slower than the pure MILP model, it appears to be more robust than the MILP model in terms of the number of search nodes required, as was also the case in the example of Section 4.8.1. It is worth noting that Thorsteinsson and Ottosson (2001) used CPLEX 7.0 to solve the MILP model (4.6) and it managed to solve only 3 out of the above 10 instances with fewer than 100,000 search nodes. The average number of search nodes explored by CPLEX 7.0 over the 3 solved instances was around 77,000.

4.8.3 Job Scheduling on Parallel Machines

We now describe the SIMPL code to implement the Benders decomposition approach of Section 4.7 (model (4.18)) when the master problem is given by (4.14), the subproblem is (4.15), and the Benders cuts are of the form (4.16).

```
01. OBJECTIVE
02.   min sum i,j of c[i][j] * x[i][j];
03. CONSTRAINTS
04.   assign means {
05.     sum i of x[i][j] = 1 forall j;
06.     relaxation = { ip:master } }
07.   xy means {
08.     x[i][j] = 1 <=> y[j] = i forall i, j;
09.     relaxation = { cp } }
10.   tbounds means {
11.     r[j] <= t[j] forall j;
12.     t[j] <= d[j] - p[y[j]][j] forall j;
13.     relaxation = { ip:master, cp } }
14.   machinecap means {
15.     cumulative({ t[j], p[i][j], 1 } forall j | x[i][j] = 1, 1) forall i;
```

Table 4.4: Job scheduling: long processing times.

		MILP		Benders		
Jobs	Machines	Nodes	Time (s)	Iterations	Cuts	Time (s)
3	2	1	0.00	2	1	0.00
7	3	1	0.02	12	14	0.09
12	3	11060	16.50	26	37	0.58
15	5	3674	14.30	22	31	0.96
20	5	159400	3123.34	30	52	3.21
22	5	$>$ 5.0M	$>$ 48h	38	59	6.70

```
16.     relaxation = { cp:subproblem, ip:master }
17.     inference = { feasibility } }
18. SEARCH
19.   type = { benders }
```

The keywords `ip:master` that appear in the relaxation statements in lines 06, 13 and 16 indicate that those constraints are to be relaxed into an Integer Programming relaxation, which will constitute the master problem. Constraints that have the word `cp` in their relaxation statements will be posted to a CP relaxation and are common to all Benders subproblems. This is true for the `xy` and `tbounds` constraints. For the `machinecap` constraints, the keywords `cp:subproblem` in line 16, together with the `forall i` statement in line 15, indicate that, for each i, there will be a different CP subproblem containing the corresponding cumulative constraint, in addition to the common constraints mentioned above. Finally, we tell SIMPL that the cumulative constraints should generate the feasibility-type Benders cuts (4.16) in line 17.

For our computational experiments, we used the instances proposed by Jain and Grossmann (2001) and, additionally, we created three new instances with more than 20 jobs. These are the last instance in Table 4.4 and the last two instances in Table 4.5. Although state-of-the-art MILP solvers have considerably improved since Jain and Grossmann's results were published, the largest instances are still intractable with the pure MILP model (4.12). In addition to being orders of magnitude faster in solving the smallest problems, the integrated Benders approach can easily tackle larger instances as well. After 48 hours of CPU time and more than 5 million branch-and-bound nodes, the best solution found by the MILP model to the last instance of Table 4.4 (22 jobs and 5 machines) had value 176, whereas the optimal solution has value 175. As noted by Jain and Grossmann (2001), when processing times are shorter the problem tends to become easier, and we report the results for shorter processing times in Table 4.5. Even in this case, the MILP model is still much worse than the integrated Benders approach as the problem size grows. For instance, after 16.9 million search nodes the MILP solver could not find a single feasible solution to the problem with 22 jobs and 5 machines. As for the largest problem (25 jobs and 5 machines), the best solution found by the MILP solver after 48 hours and 4.5 million search nodes had value 181, and the optimal value is 179.

4.9 Final Comments and Conclusions

In this paper we describe a general-purpose integrated solver for optimization problems, to which we refer by the acronym SIMPL. The main idea behind SIMPL is to note that many of the traditional

Table 4.5: Job scheduling: short processing times.

		MILP		Benders		
Jobs	Machines	Nodes	Time (s)	Iterations	Cuts	Time (s)
3	2	1	0.00	1	0	0.00
7	3	1	0.01	1	0	0.01
12	3	4950	1.98	1	0	0.01
15	5	14000	19.80	1	0	0.03
20	5	140	5.73	3	3	0.12
22	5	$>$ 16.9M	$>$ 48h	5	4	0.38
25	5	$>$ 4.5M	$>$ 48h	16	22	0.86

optimization techniques can be seen as special cases of a more general solution approach which iterates over three steps: solving relaxations, performing logical inferences and intelligently enumerating problem restrictions.

We ran computational experiments on three different classes of optimization problems, comparing the modeling effort and performance of a commercial MILP solver (CPLEX) relative to SIMPL. Our results illustrate at least three main benefits of a general-purpose integrated solver: (a) it solves two of the problem classes faster, one of them by orders of magnitude, even though it is still an experimental code; (b) it presents more robust behavior across different instances of a problem class; and (c) it provides these advantages without increasing the modeler's burden, since the integrated models are written in a concise and natural way.

One may argue that it is unfair to compare SIMPL with an off-the-shelf commercial solver, since the latter does not contain facilities to exploit problem structure in the way that SIMPL does. Yet a major advantage of an integrated solver is precisely that it can exploit structure while remaining a general-purpose solver and providing the convenience of current commercial systems. SIMPL's constraint-based approach automatically performs the tedious job of integrating solution techniques while exploiting the complementary strengths of the various technologies it combines.

Our examples suggest that a SIMPL user must be more aware of the solution algorithm than a CPLEX user, but again this allows the solver to benefit from the user's understanding of problem structure. We anticipate that future development of SIMPL and related systems will allow them to presuppose less knowledge on the part of the average user to solve less difficult problems, while giving experts the power to solve harder problems within the same modeling framework. In addition, we plan to increase SIMPL's functionality by increasing its library of metaconstraints, solver types, constraint relaxations, and search strategies, with the goal of accommodating the full spectrum of problems described in Table 4.1.

Chapter 5

Building Efficient Product Portfolios at John Deere

Joint work with: Dominic Napolitano, Alan Scheller-Wolf and Sridhar Tayur.

Abstract: John Deere & Company (Deere), one of the world's leading producers of machinery, manufactures products comprised of various *features*, within which a customer may select one of a number of possible *options*. On any given Deere product line there may be tens of thousands of combinations of options – configurations – that are feasible. Maintaining such a large number of configurations inflates overhead costs; consequently, Deere wishes to reduce the number of configurations from their product lines without upsetting customers or sacrificing profits. In this paper we provide a detailed explanation of the marketing and operational methodology used, and tools built, to evaluate the potential for streamlining two product lines at Deere. We illustrate our work with computational results from Deere, highlighting important customer behavior characteristics that impact product line diversity. For the two *very different* studied product lines, an increase in profit from 8-18% has been identified, possible through reducing the number of configurations by 20-50% from present levels, while maintaining the current high customer service levels. Based on our analysis and the insights it generated, Deere recently implemented a new product line strategy. We briefly detail this strategy, which has thus far increased profits by tens of millions of dollars.

5.1 Introduction

Deer & Company (Deere) manufactures equipment for construction, commercial, and consumer applications, as well as engines and power train components. As a major player in many equipment markets, Deere maintains multiple product lines; within each line, there may be several thousand, to several million, different product variants. Variants are built by selecting, for each *feature* available on a machine – e.g. engine type, transmission and axle – one of a number of possible *options* – e.g. 200, 250 or 300 horsepower (HP) for engines. Not all options are compatible; a feasible combination of options is called a *configuration*.

Deere speculates that maintaining too many configurations reduces profits, by elevating what Deere calls *complexity cost*. This cost, over and above the inventory carrying costs of each configuration, captures factors such as reduced manufacturing efficiency, frequent line changeovers, and the general overhead of maintaining documentation and support for a configuration. This definition is similar to that given in Thonemann and Brandeau (2000), where complexity cost is, "the cost of indirect

functions at a company and its suppliers that are caused by component variety; complexity cost includes, for instance, the cost of designing, testing, and documenting a component variant." In this paper we describe the marketing and operational methodology and tools we developed to reduce Deere's complexity costs, by concentrating product line configurations while maintaining high customer service, thus elevating overall profits. We illustrate our work with applications to two lines at Deere; details of the products have been disguised, but the lines differ in significant ways (costs, profits, sales), making them a diverse test bed for our optimization algorithm.

A primary component in our algorithm is our *customer migration model*, quantifying the behavior of Deere's customers: A customer may want a specific configuration, but if his/her first choice is unavailable he/she may *migrate* to an alternative configuration that does not differ too greatly from the first choice. Using actual sales, along with customer segmentations and part worths utilities provided by Deere, we probabilistically model every customer as *individually* identifying a set of acceptable configurations, sorted in decreasing order of preference: their *migration list*. When the top configuration on his/her list is not available, a customer will buy the next available configuration. When no configuration is available, the customer *defects* to a competitor.

Using each customer's migration list, as well as costs and profits for all feasible configurations (found via Constraint Programming, Marriott and Stuckey, 1998), we build a Mixed Integer Program to maximize Deere's profits within a product line. Application of our algorithm provides: (i) A general method to determine which configurations are least profitable, and thus may be candidates for elimination; (ii) Recommendations for how Deere can significantly focus specific product lines; and (iii) Identification of the high level drivers of product line efficiency. Based on our results, Deere has instituted an incentive program to steer customers towards a core subset of their product lines, which has resulted in increased profit in line with our predictions – tens of millions of dollars annually (see Section 5.7 for details).

We begin with a review of the literature, in Section 5.2. We then describe how we generate and cost out all of the feasible configurations on a product line, in Section 5.3. Next we detail how we generate customer utilities and migration lists, in Section 5.4. The development and solution of our Mixed Integer Program is related in Section 5.5. We present results of the experiments on the Deere product lines in Section 5.6, briefly describe Deere's actual implementation in Section 5.7, and conclude by summarizing our work and findings in Section 5.8.

5.2 Literature Review

Reducing product line complexity is a common goal among companies. For example, Raleigh (2003) described how Unilever uses its *Product Logic* framework to simplify its global Home and Personal Care product portfolio. Similarly, Stalk Jr. and Webber (1993), Henkoff (1995), and Schiller (1996) describe pruning of product lines in several other industries. From a marketing perspective, there are often fears that such a reduction may detract from brand image or market share (Kahn, 1995; Chong, Ho, and Tang, 1998; Kekre and Srinivasan, 1990), but there are also works, like Randall, Ulrich, and Reibstein (1998), that discuss the dangers of extending a product line. These latter concerns are in keeping with findings that *reducing* the breadth of lines and focusing resources on popular products, "favorites", may actually *increase* sales, see Quelch and Kenny (1994); Broniarczyk, Hoyer, and McAlister (1998); and Fisher, Jain, and MacDuffie (1995). Our model is consistent with both of these streams of thought: If a customer finds a product that meets their needs (i.e. a "favorite") he or she will make a purchase; if such a product is no longer part of the product line they will not (and market share will go down).

The motivation behind reducing product lines is often cost containment; there is a long history

of tying product line complexity to increased costs. This has been done analytically by Hayes and Wheelright (1984); Abeggeln and Stalk (1985); and Kekre (1987); and empirically, by Foster and Gupta (1990); Anderson (1995); and Fisher and Ittner (1999). Deere has spent considerable time prior to this project establishing the linkage between variety and cost, ultimately resulting in a proprietary function linking the two, the "complexity cost".

Due to the ubiquitous and cross-functional nature of product line optimization, interdisciplinary research has long been advocated; Yano and Dobson (1998b) and Ramdas (2003) provide an extensive literature survey, which we briefly summarize. McBride and Zufryden (1988), Kohli and Sukumar (1990) and Nair, Thakur, and Wen (1995) use math programs to solve the product portfolio problem, but do *not* include costs related to the breadth of the line, as this is fixed. Kohli and Sukumar (1990) apply their method to data from a real (but small) problem for telephone hand sets. Green and Krieger (1985), and Dobson and Kalish (1988, 1993) present math programming formulations and solution heuristics to the product selection and pricing problem, which *do* include product line breadth costs. More generally, De Groote (1994), Raman and Chhajed (1995) and Yano and Dobson (1998a) use iterative solution procedures to determine attributes, prices, and production processes, each having a specific fixed cost. Morgan, Daniels, and Kouvelis (2001) solve a similar problem using a mathematical program. None of these algorithms are suitable for problems anywhere near the size of Deere's. Likewise none of these works include applications to any actual industrial data. Recently there have been alternative formulations: Chen, Elishberg, and Zipkin (1998), and Chen and Hausman (2000), again tested on very small, non-industrial data.

Thus, while the problem of determining optimal product lines has a long and important place within the marketing and operations literatures, actual *industrial scale* applications of analytical techniques to *optimize* product lines based on *detailed* company data have never been performed. We provide these for the first time.

5.3 Building and Pricing Feasible Configurations

In this section we describe the generation of all feasible product configurations, their unit costs, and profits. This is no trivial task, as machines can have as many as 23 features (engines, transmissions, attachments) and multiple options within each feature (tires may have 25 options). In principle, this could create millions of possible configurations. In practice, however many combinations of options are not physically possible; for example, certain transmissions cannot be used with certain axles. Consequently every product line has a unique set of *configuration rules* that determine how the line's options can be combined. These rules typically take the form "if A is true then B must be true", where A and B may be complicated sequences of conjunctions, negations, and disjunctions. There are 39 combination rules for the Gold Line, and 68 for the Silver. There are also *pricing rules* (35 for the Gold Line and 212 for the Silver), which determine the total cost of a machine given a choice of options. Once the price and total cost for a specific configuration are known, its profit contribution can be computed. We use Eclipse (Wallace et al., 1997), a Constraint Programming (CP) language (Tsang, 1993; Marriott and Stuckey, 1998) to build, cost out and find the profit for all of the valid configurations, leveraging CP's inherent expressive power and intelligent search mechanisms.

We provide examples of configuration and pricing rules below, and then illustrate how the first of these rules would be incorporated as a constraint into our CP formulation.

Sample configuration rules:

1. Feature MidPTO unavailable on machines with both a 2WD axle and SST transmission;
2. For engine 32_130DLV and transmission PRT, axle cannot be 2WD.

Sample pricing rules:

1. If engine type is 28_129DLV and the machine does not have a MidPTO, SST transmission costs $1000 for a 2WD axle and $1200 for a 4WD axle;
2. If engine type is 32_130DLV, MidPTO costs $250 in two circumstances: either the axle is 4WD, or the axle is 2WD and the transmission type is not SST.

Configuration Rule Example: Let the line under consideration have n different features. A vector x of n variables $x_1, \ldots, x_n$ represents an option choice for each of the n features; if feature i has m_i different options, the domain of x_i is defined as $\{1, \ldots, m_i\}$. We write all combination and pricing rules as constraints on the x variables. For instance, if transmission is the second feature and SST is its third option for transmissions; axle is the third feature and 2WD is its first option; and MidPTO is the fifth feature and it is present when $x_5 = 1$, then configuration rule 1 above is:

$$x_2 = 3 \wedge x_3 = 1 \Rightarrow x_5 \neq 1.$$

After all necessary constraints are added to the CP model, the solver outputs all feasible assignments of values to the x variables (i.e. all feasible configurations), as well as their prices.

5.4 The Customer Migration Model

When Deere approached us they did not have a specific customer behavior model in mind, but they did know which product features were important, and what types of trade-offs customers were likely to make. We were tasked with giving structure to this knowledge; we were to provide a tool to harness Deere's understanding, incorporate customer purchase data (over fifteen thousand data points), and be well-suited for use by our optimization engine. We were *not* tasked with evaluating customer behavior, for example by estimating utility values; rather we were to provide a framework for Deere to use, now and in the future, based on *their* understanding of customer behavior.

In order to accomplish this task, we first sought a high-level view of how a typical Deere customer behaves. We posited the following model: If provided with the entire selection of Deere products, the customer could, and would, rank them in a mental list in order of preference, based on the different options each configuration possessed. If money were no object the customer would purchase the configuration at the top of their list, if it were available. If it were too expensive, or not available, the customer would move down their list until they found an available, and affordable, machine. At some point, if too many of their top choices were either unavailable or exceeded their budget, a customer would give up and leave the Deere dealer without making a purchase.

This idea formed the basis for our construct capturing customer behavior: the *customer migration list*. To operationalize this model, guided by Deere's marketing research, we generated a set of *randomized* migration lists for each product line – one list for each purchase from Deere's sales history. The lists are generated prior to solving our Mixed Integer Program, to increase solution speed and allow study of the lists in their own right. This also facilitates experimental replication and sensitivity analysis (see Section 5.6).

What remains is to explain how we model customers' formation of migration lists – which configurations they will consider (their *potential configurations*), how they evaluate these potential configurations, decide what is "too much to pay", and at what point give up and leave. In Section 5.4.1 we discuss how (and why) we probabilistically place customers into different segments, how we then specify key parameter values (contingent on segment), and determine the customer's potential configurations – those configurations that could *possibly* satisfy the customer's wants. These are then

ranked, according to "part worths utilities", as described in Section 5.4.2. The final migration list is formed according to these rankings, with probabilistic cut-offs based on price and utility. This is described, along with a complete summary of our generation algorithm, in Section 5.4.3. We finally discuss extensions of our migration list algorithm in Section 5.4.4.

Throughout, when discussing the generation of the migration list of an individual customer, C_0 will denote the configuration this customer purchased, with price P_0, and summed part worths utilities equal to U_0.

5.4.1 Segmenting Customers and Determining Parameter Values

Our first step was to segment customers, to more accurately capture market behavior. While this is not mandatory for our algorithm, Deere felt that there were a number of distinct, but overlapping customer segments for their products – three for the Gold line and four for the Silver. (Different segments correspond, for example, to private users and commercial users in different industries.) Since different segments make purchases from the same, single product line, they must be considered simultaneously. Below we define the parameters that place customers into segments, and then the parameters that determine potential configurations for customers *within each segment*.

The most direct way to place a customer into a segment is to match the options on his/her purchased configuration to a segment likely to purchase it. If options on a configuration are attractive to different segments, then the configuration will have to be assigned in some probabilistic manner. Moreover, for those configurations that are not (probabilistically) assigned there will have to be a procedure to place these into segments as well. These ideas are formalized in the two segmentation parameters below, which are defined for each segment. We then present an illustrative example.

(cond, q) – Based on whether certain options denoted by the logical condition **cond** (e.g. "engine is 400 HP and axle is 4WD") are present in C_0, the customer is placed in the segment with probability q (the proportion of purchases with these options from that segment). Multiple segments for which C_0 satisfies **cond** are considered sequentially, until the customer is assigned. Should a customer fail to be placed, she/he is assigned according to s, below;

s – This is the percentage of unit sales that come from the segment. All customers not assigned according to the rule above are randomly assigned based on these probabilities. The sum of s values over all segments is thus equal to one.

Customer Segmentation Example: Assume there are three customer segments with the following parameters:

Segment	cond	q	s
1	engine = 450HP	.9	.2
2	axle = 2WD and transmission = SST	.75	.55
3	stereo = 6-disc CD	.2	.25

Table 5.1: Segments for Customer Segmentation Example.

Assume configuration C_1 has a 450HP engine, 4WD transmission, and a 6-disc CD stereo. It will be placed in segment 1 with 90% probability. If this does not happen segment 2 will be bypassed, as C_1 is not 2WD. Since C_1 has a 6-disc stereo, it will then be tried in segment 3, and assigned there with a 20% probability. If C_1 is not being assigned after these attempts, it will be assigned to one of the three segments with probabilities equal to 20%, 55% and 25%, respectively.

After placing the customer into a segment we assign values to five key parameters – proxies for the critical elements of customer behavior. Because these behaviors vary across segments, they are defined for each segment. Moreover, in order to capture heterogeneity within segments, many of the parameters are either probabilities themselves, or are selected probabilistically. Taken together, these five parameters determine a customer's potential configurations; they describe how closely the elements of the migration list will hew to the purchased configuration, C_0, that spawned it. The first two, c and $\mathcal{F}$, determine the size of the search space a customer is willing to consider in the neighborhood of C_0. The next two, u and r, determine how flexible the customer is willing to be within this search space, with respect to price and total part worths utility of configuration. The final parameter, β, controls whether or not C_0 will be forced to the top of the migration list. Values for each of these parameters, by segment, were determined by Deere.

c – This is the *commonality factor*, the minimum number of options a configuration must have in common with C_0 in order for the customer to be willing to purchase it. If $c = 0$, all machines may be considered; as c grows, fewer deviations from C_0 are allowed, until (when c is maximal), no configuration except C_0 will be considered.

$\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ – This is a list (possibly empty) of fixed features. Any feature that appears in this set must not have its option changed from that on C_0.

r – This models customers' reservation prices. A customer who purchased a machine with price P_0 has his or her reservation price drawn uniformly from the interval $[P_0, (1+r)P_0]$. Customers are willing to buy machines costing at most their reservation price.

u – This models customers' reservation utilities. Analogous to r, a customer who purchased a machine with total part worths utility U_0 has his or her reservation utility drawn uniformly from the interval $[(1-u)U_0, U_0]$. Customers are willing to buy machines with total part worths utility no less than their reservation utility.

β – This is the *first-choice* probability. With probability β, C_0 will be placed first on the customer's migration list, independent of the utilities generated by the model. With probability $1-\beta$, the position of C_0 on the list will be determined by its utility. If $\beta = 0$ we have a pure choice model, and if $\beta = 1$ customers always prefer their initial purchase.

After a customer is placed into a segment and the above parameters are established we use constraint programming to determine the customer's set of potential configurations: Those configurations having at least c features in common with C_0, including all of those on list $\mathcal{F}$, and price no more than the customer's reservation price. The next step is to determine how the customer will evaluate these configurations, how he/she will rank them on his/her migration list.

5.4.2 Using Option Utilities to Rank Configurations

To model customers' ranking of configurations we utilize part worths utilities; in a standard application of part worth utilities the (static) value of all of the options on a configuration would be summed, to give the total value of the configuration. We randomize this procedure to allow for greater heterogeneity between customers within the same segment. This randomization is crucial to our algorithm; without it customers in the same segment who purchased the same configuration would have identical, or nearly identical, migration lists; they could differ only in where they end, due to reservation price, and whether C_0 was first, if $\beta \neq \{0, 1\}$.

As is standard, within each feature (for example engine) we initially assign values determined by Deere, by segment, to all available options (for example 200HP, 400HP). This also defines the feature's mean *relative importance* – the difference between the largest and smallest utility value for any of the its options – which captures the feature's baseline importance to customers in the segment. For each customer we then randomly perturb these mean relative importance values using *deviation* parameters, again estimated by Deere. This captures the fact that individual customers from the same segment may vary considerably in how value different features. (We could have defined sub-segments to capture this phenomenon, but given the data available – heterogeneity is known to be present but is hard to quantify – we preferred the current method.)

To perturb relative importance values our algorithm randomly selects a fixed number of features and rescales their relative importances, using a factor uniformly drawn from the feature's mean relative importance plus or minus its deviation parameter. We then rescale the relative importances of those features *not* selected, to keep the sum of all the relative importance parameters constant. Next the option utilities within each feature are rescaled by the ratio of the new relative importance value to the old. Only then do we sum the individual part worths utilities, yielding the total value for each configuration. This operation is done uniquely for all customers within the segment. We formalize this procedure below, and then provide an example.

1. Randomly select n features, where n is a fixed parameter for all customers, in all segments;
2. Randomly generate a relative importance for each of the n features using a uniform distribution over the feature's mean relative importance, plus or minus its deviation;
3. Determine the relative importance of the remaining features by scaling them such that the total sum of relative importances has the same value as it had before step 2;
4. Rescale all option utilities based on the new relative importances of their features.

Utility Calculation Example: We consider a simplified machine with three features – engine, transmission and axle. For a given customer segment, the first section of Table 5.2 presents coded options available in each feature, with their corresponding initial utilities. The relative importance of the three features (mean, deviation) are (21, 5), (20, 3) and (15, 4), respectively.

Engine		Transmission		Axle	
Option	Utility	Option	Utility	Option	Utility
Initial Utilities					
200HP	0	MPT	0	AWD	0
250HP	7	HWT	10	NWD	15
300HP	14	LBT	20		
350HP	21				
Perturbed Utilities					
200HP	0	MRT	0	AWD	0
250HP	6.61	HWT	11	NWD	14.17
300HP	13.22	LBT	22		
350HP	19.83				

Table 5.2: Initial and perturbed utilities for Utility Calculation Example.

The sum of the mean relative importances is 56. Assume $n = 1$ and transmission was selected to have its relative importance changed to 22 (steps 1 and 2). Thus the new relative importances

of engine and axle have to sum to 34 (= 56 − 22); the relative importance of engine becomes 19.83 (= $\frac{21}{21+15} \times 34$), and of axle 14.17 (= $\frac{15}{21+15} \times 34$) (step 3). We then rescale the option utilities by $\frac{\text{new RI of feature}}{\text{old RI of feature}}$ (step 4). The lower portion of Table 5.2 shows the results.

5.4.3 Formation of the Final Migration List

After we have obtained all of a customer's potential configurations (as described at the end of in Section 5.4.1), and determined their part worths utilities (as outlined in Section 5.4.2), we form the customer's final migration list. We list all of the potential configurations in decreasing order of their summed part worths utilities, stopping when we reach a configuration with summed utility less than the customer's reservation utility. More formally, for every customer in the sales history, we generate a migration list as follows (recall that C_0 is the configuration bought by the customer):

1. Assign the customer to a segment as described in Section 5.4.1; this determines c, $\mathcal{F}$ and β;
2. Determine reservation price and reservation utility for the customer, as defined in Section 5.4.1;
3. Determine customer's potential configuration list – all configurations that have at least c features in common with C_0, including all those on list $\mathcal{F}$, with price no more than the reservation price. Call this list L;
4. Determine the option utility values for the customer as in Section 5.4.2;
5. Sort L in decreasing order of summed part worths utilities, truncating the list when a configuration has utility value less than the reservation utility;
6. Use the parameter β to determine whether to move C_0 to the top of L;
7. Optionally, truncate L, ensuring that C_0 remains in L after truncation (in accordance with wishes of Deere);
8. Output the resulting list L.

5.4.4 Summary and Extensions

Our algorithm generates a set of lists; each list models an individual customer's choices, while in aggregate the lists capture behavior over segments. Customer heterogeneity is captured in two ways: Based on their initial purchases, different customers consider different potential configurations. Within these potential configurations, due to our randomization of features' relative importances, customers may value configurations differently.

We can use lists generated from different parameter settings to explore the sensitivity of product lines to different beliefs about customer behavior (as in Section 5.6.5). Or, if lists are generated via a different method, such as expert surveys, these could be used in our Mixed Integer Program (MIP), enabling us to compare the solutions to tune customer behavior parameters. In addition, we can optimize different random replications of the same customer behavior parameters to estimate the variation of the optimal product portfolio for the same parameter set (as in Section 5.6.3).

Our list generation algorithm could be extended to include machines from Deere's competitors, possibly with a utility for brand equity, modeling a richer competitive environment. Likewise, we could allow customers to make multiple purchases, include "lost" customers who were unable to make purchases due to factors at the dealers, and/or customers who did not find their first choice, but did buy a different machine. In the first case, we could generate multiple identical (or correlated) lists

for selected customers, based on historical purchase data. To deal with "lost" customers we could generate an additional set of customers and their initial choices, possibly using the s parameters. Finally, we could assign an explicit probability that any customer's purchase was not actually his/her first choice (although the β parameter captures this to some extent). None of these actions was used for the project reported.

One extension not readily made is to include negotiations with customers, as all prices are taken as exogenous. Endogenous pricing raises significant and complex modeling issues outside of the scope of this project. But, if data on actual sales prices were made available, profits could be rescaled if discounts were significant. In general, if Deere were to provide us with more detailed data, our algorithm should improve. Additional information which would be most helpful would focus on customer migration behavior – for example how long should customer migration lists be, should these lengths change according to segment, how elastic are reservation prices and utilities (and utilities in general), and how do incentives affect migration behavior and customer impressions. Through their implementation (see Section 5.7) Deere is gathering some of this data now.

5.5 The Optimization Model

In this section we describe the MIP model that selects the configurations to build in order to maximize total profit. Although it is in theory potentially of exponential complexity (it enumerates the search space), our MIP can efficiently solve Deere's base problem (5000-15000 customers and 3500-24000 configurations) using commercially available solvers, and also allows the incorporation of additional managerial constraints. Examples of these include limits on the number of configurations built, a minimum service level, and specifying configurations that must be included (or dropped). The performance of our algorithm with some of these constraints is shown in Section 5.6.6.

5.5.1 Input Data and Decision Variables

The input for the optimization problem consists of the following data.

- T — The set of all relevant configurations. $T = \{1, 2, \ldots, t\}$;
- C — The set of all customers;
- L_i — The ordered set of configurations in the migration list of customer i, $\forall\ i\ \in\ C$;
- N_j — The set of customers whose migration lists contain j: $N_j = \{i \in C \,|\, j \in L_i\}$;
- p_j — Profit (price − cost) of configuration $j \in T$.

Our model uses two sets of binary decision variables:

- $y_j = 1$ if configuration j is produced, 0 otherwise ($j \in T$);
- $x_{ij} = 1$ if customer i buys configuration j, 0 otherwise ($i \in C$, $j \in L_i$). Determining the y_j values likewise determines the x_{ij} values, given a set of migration lists.

5.5.2 Constraints

Our model uses the following constraints:

$$x_{ij} \leq y_j, \qquad \forall\, j \in T,\ i \in N_j; \tag{5.1}$$

$$\sum_{k \text{ after } j \text{ in } L_i} x_{ik} + y_j \leq 1, \qquad \forall\, j \in T,\ i \in N_j; \tag{5.2}$$

$$y_j \in \{0,1\}, \qquad \forall\, j \in T; \tag{5.3}$$

$$x_{ij} \in \{0,1\}, \qquad \forall\, i \in C,\ j \in L_i. \tag{5.4}$$

Constraints (5.1) prescribe that only available configurations can be purchased, and (5.2) that configurations that appear earlier in a migration list of a customer have higher priority. Note that (5.2) also ensure that every customer buys at most one configuration.

5.5.3 Objective Function

Deere wishes to maximize the following objective, which is similar to many in the literature, (for example it is a more detailed version of that in Dobson and Kalish 1988, 1993):

$$\max \ \text{Shareholder Value Added (SVA)} = \sum_{j \in T} \left((p_j - K) \sum_{i \in N_j} x_{ij} \right) - a - \mathit{Overhead}\,. \tag{5.5}$$

Overhead is the sum of fixed costs, and the constant K captures the inventory and capital equipment costs incurred for each unit sold. Reducing the number of configurations in the product line will *not* reduce K, only selling fewer units will. Thus K differs from complexity cost, a, which *does* depend on the number of configurations in the product line via (5.6)-(5.10) below.

Returning first to K, this is defined as

$$K = \text{Wacc}\left(\delta P_{\text{avg}} + \lambda S_{\text{avg}} + \text{PPE}\left(1 + \frac{g}{\text{Wacc}}\right)\right),$$

where the following constants were provided by Deere:

Wacc – Weighted average cost of capital;

δ – finished goods inventory to sales ratio;

$P_{\mathbf{avg}}$ – average value of a unit of finished goods inventory;

λ – raw inventory to sales ratio (number of components and spare parts in inventory per unit sold);

$S_{\mathbf{avg}}$ – average value of a part in raw inventory;

PPE – ratio of Net Plant Property and Equipment to volume of units sold;

g – ratio of depreciation expense to Net Plant Property and Equipment;

The term a in (5.5) is our piecewise-linear approximation of Deere's complexity cost function. Use of this cost form dates from before the initiation of our project; its determination and/or validation was outside of the scope of our work. We discuss the use of an alternative, and potentially more accurate complexity cost function in equations (5.11) - (5.13).

To define a, we need an extra set of variables and constraints. Let B be the set of grid points we choose to approximate the complexity function. For every $k \in B$, let n_k and o_k be, respectively, the number of configurations and Deere's complexity cost corresponding to the k^{th} grid point. Finally,

let λ_k be real-valued variables that determine a convex combination of two consecutive grid points. Then, using SOS2, a device available in most commercial MIP solvers that ensures at most two of its arguments assume positive values (and if two arguments are positive they are consecutive):

$$\sum_{j \in T} y_j = \sum_{k \in B} n_k \lambda_k, \tag{5.6}$$

$$a = \sum_{k \in B} o_k \lambda_k, \tag{5.7}$$

$$\sum_{k \in B} \lambda_k = 1, \tag{5.8}$$

$$\text{SOS2}(\lambda_1, \ldots, \lambda_{|B|}), \tag{5.9}$$

$$\lambda_k \in [0, 1], \qquad \forall\, k \in B\,. \tag{5.10}$$

An alternative formulation of a complexity cost could use a step function over B. Let z_i ($i \in \{1, \ldots, |B|\}$) be a binary variable indicating whether the number of configurations is at least n_i. Setting $o_{\text{zero}} = 0$, we modify the previous objective function by substituting $\sum_{i=1}^{|B|-1}(o_i - o_{i-1})z_i$ for a in (5.5). To complete this new formulation, we drop (5.6)–(5.10) and include (5.11)–(5.13).

$$\sum_{j \in T} y_j \leq \sum_{i=1}^{|B|-1} (n_{i+1} - n_i) z_i + n_1, \tag{5.11}$$

$$z_{i+1} \leq z_i\,, \qquad \forall\, i \in \{1, \ldots, |B| - 2\}; \tag{5.12}$$

$$z_i \in \{0, 1\}\,, \qquad \forall\, i \in \{1, \ldots, |B| - 1\}. \tag{5.13}$$

We illustrate the performance of our algorithm under this objective function in Section 5.6.6.

5.5.4 Properties of Optimal Solutions

Proposition 2 *When looking for optimal solutions, imposing integrality on all the y variables alone implies integrality of all the x variables, and vice-versa.*

Proof. Let $y_j \in \{0, 1\}$ for all $j \in T$. Let $Y_0 = \{j \in L_i \,|\, y_j = 0\}$ and $Y_1 = \{j \in L_i \,|\, y_j = 1\}$, the sets of configurations on the list which are not, and are, produced, respectively. Constraints (5.1) say that $x_{ij} = 0$ for all $j \in Y_0$. For every customer $i \in C$, let $j_i \in Y_1$ be the produced configuration that appears earliest in L_i. If configuration k comes before j_i in L_i, (5.1) makes $x_{ik} = 0$. If configuration k comes after j_i in L_i, (5.2) makes $x_{ik} = 0$. Finally, x_{ij_i} will be equal to 1 due to the objective function (here we assume all configurations have positive profit). If no such j_i exists for a given customer i, then all x variables for customer i are set to zero because of (5.1).

Now let $x_{ij} \in \{0, 1\}$ for all $i \in C$ and $j \in L_i$. Let $Y_2 = \{j \in T \,|\, x_{ij} = 1 \text{ for some } i \in C\}$ be those configurations that have to be produced. Constraints (5.1) will force $y_j = 1$ for all $j \in Y_2$. Also, (5.2) will make $y_j = 0$ whenever there exists a customer $i \in C$ such that $x_{ik} = 1$ and j precedes k in L_i. Finally, all remaining y variables that have not been fixed will be set to zero because we are maximizing, and a is a non-decreasing function of the sum of all y variables. □

Proposition 3 *At least one of the integrality conditions (5.3) or (5.4) is necessary to avoid the possibility of fractional optimal solutions.*

Proof. Let us consider an example in which constraints (5.3) and (5.4) are relaxed, i.e. $\{0,1\}$ is replaced by $[0,1]$. Let $C = \{1,2,3\}$, $T = \{1,2,3\}$, $L_1 = \{1,2\}$, $L_2 = \{2,3\}$ and $L_3 = \{3,1\}$. In addition, let $p_1 - K = 101$, $p_2 - K = 103$, $p_3 - K = 102$, $Overhead = 0$, and let the complexity cost of producing 1, 2 or 3 configurations be 200, 250 and 300, respectively. If we decide to build only one configuration, the best choice is to make $y_2 = 1$, which yields a profit of $2 \times 103 - 200 = 6$. If we decide to build two configurations, the best choice is to make $y_2 = y_3 = 1$, which yields a profit of $2 \times 103 + 102 - 250 = 58$. Finally, if we make $y_1 = y_2 = y_3 = 1$, our profit is equal to $101 + 103 + 102 - 300 = 6$. But, if we make $y_1 = y_2 = y_3 = 0.5$, we will have maximal profit of $101 + 103 + 102 - 225 = 81$. This gives us an integrality gap of $(81 - 58)/58 \approx 39.7\%$. □

5.5.5 Strengthening the MIP Formulation

Let $\mathcal{P}$ be the polytope defined as the convex hull of integer points satisfying the system of constraints (5.1)–(5.4). Proposition 3 shows that (5.1) and (5.2), together with the linear relaxation of (5.3) and (5.4), are not sufficient to describe $\mathcal{P}$. In this section we present a brief polyhedral study of $\mathcal{P}$, which yields some additional valid inequalities for it.

Let $\mathcal{P}_H$ be a higher dimensional representation of $\mathcal{P}$ including new binary variables $\overline{y}_j$ satisfying the constraints $y_j + \overline{y}_j = 1$, for every $j \in T$. (Following our terminology, $\overline{y}_j = 1$ if and only if configuration j is *not* produced.) In addition, let us rewrite (5.1) as $x_{ij} + \overline{y}_j \leq 1$ in the description of $\mathcal{P}_H$. The *submissive polytope* $\mathcal{P}_H^{\leqslant}$ of $\mathcal{P}_H$ can be obtained from $\mathcal{P}_H$ by substituting '$\leq$' for '$=$' in every constraint of the form $y_j + \overline{y}_j = 1$. It is easy to see that $\mathcal{P}_H^{\leqslant}$ is a vertex packing polytope (see Padberg 1973). Then, we can associate an undirected graph G to $\mathcal{P}_H^{\leqslant}$: variables correspond to vertices of G, and left hand sides of constraint of $\mathcal{P}_H^{\leqslant}$ correspond to cliques in G.

Definition 2 *A* preference cycle X *is a sequence of variables* $x_{i_1 j_{11}}$, $x_{i_1 j_{12}}$, $x_{i_2 j_{21}}$, $x_{i_2 j_{22}}$, ..., $x_{i_k j_{k1}}$, $x_{i_k j_{k2}}$, *such that* $k \geq 2$, $j_{p2} = j_{(p+1)1}$ *for all* $p \in \{1, \ldots, k-1\}$, $j_{k2} = j_{11}$, *and, for every customer* $i \in \{i_1, \ldots, i_k\}$, j_{i1} *precedes* j_{i2} *in* L_i. *Let* $D(X) = \{j_{12}, j_{22}, \ldots, j_{k2}\}$ *be the set of less desirable configurations in the cycle.*

Given a preference cycle X, let $G[X]$ be the subgraph of G induced by the variables (vertices) in X as well as all variables y_j and $\overline{y}_j$ such that $j \in D(X)$. By construction, the vertices of $G[X]$ corresponding to y_j, $\overline{y}_j$ and x_{ij} for $j \in D(X)$ form a chordless cycle (or *hole*) in $G[X]$. When k is odd, Padberg (1973) showed that the following inequality is facet-defining for the vertex packing polytope defined over $G[X]$.

$$x_{i_1 j_{12}} + x_{i_2 j_{22}} + \cdots + x_{i_k j_{k2}} + \sum_{j \in D(X)} (y_j + \overline{y}_j) \leq \left\lfloor \frac{3k}{2} \right\rfloor. \tag{5.14}$$

Clearly, (5.14) is also valid for $\mathcal{P}_H$ and, since we have $y_j + \overline{y}_j = 1$ in $\mathcal{P}_H$, (5.14) can be rewritten as:

$$x_{i_1 j_{12}} + x_{i_2 j_{22}} + \cdots + x_{i_k j_{k2}} \leq \left\lfloor \frac{k}{2} \right\rfloor. \tag{5.15}$$

As an example, if we consider the preference cycle in the proof of Proposition 3, the fractional solution presented there is excluded by the inequality $x_{12} + x_{23} + x_{31} \leq 1$. Inequalities (5.14) are not necessarily facet defining for $\mathcal{P}_H^{\leqslant}$, but in some cases, they can be lifted to become facet defining (see Padberg 1973 for more details).

As shown in Section 5.6, our initial formulation already presents satisfactory results in terms of computational performance. Therefore, we decided not to implement a branch-and-cut algorithm using the above inequalities (or strengthenings thereof). Nonetheless, such inequalities may prove

useful in larger problem instances. Nemhauser and Sigismondi (1992) give details about implementing a branch-and-cut algorithm for vertex packing, and Grötschel, Lovász, and Schrijver (1988) give a polynomial time separation algorithm for odd hole inequalities such as (5.14).

5.6 Computational Results

In this section we report computational results from two product lines at Deere, henceforth referred to as the Gold and Silver lines. The fundamental characteristics of these lines can be gleaned from Tables 5.3-5.5: The Silver line is a higher cost, higher margin product line with fewer sales and a larger number of features and feasible configurations. Not surprisingly, the Silver line's customer base is also more heterogeneous than the Gold's – the Silver line has more segments, and the differences between how segments value features is greater. In addition, *within* segments many Silver line features are of approximately the same importance to customers. This leads to larger and more variable migration lists, seen in Table 5.4, which significantly affects optimization.

We first report in Section 5.6.1 instance sizes and representative generation times for the feasible configurations and migration lists. The subsequent subsections, 5.6.2 to 5.6.6, detail solutions of particular instances of the problems introduced in Section 5.6.1. Section 5.6.7 summarizes our findings. All execution times are expressed as CPU time of a Pentium 4, 2.3 GHz, with 2GB of RAM. Execution parameters have been disguised to maintain proprietary information.

5.6.1 Generating Feasible Configurations and Migration Lists

The first step in the optimization is to determine the number of configurations in a product line, as described in Section 5.3. As seen in Table 5.3, in all cases configuration generation was quite swift.

	Features	Feasible Configurations	Typical Generation Time (seconds)
Gold	9	3696	2
Silver	23	24144	60

Table 5.3: Features, feasible configurations and generation times.

The next step is to generate the migration lists, as outlined in Section 5.4. The commonality factor, c, plays a significant role in this process by limiting potential configurations: A smaller c allows a customer to vary more features, or be more flexible with respect to what he/she will buy, reducing the weight the algorithm places on C_0. We allow customers on the Gold and Silver lines to vary two and three features, respectively, from their C_0 (we experimented with this – see Section 5.6.5). In addition to making customers more true to their purchases, thus preserving heterogeneity, these restrictions had the practical effect of keeping list-generation times manageable, as seen in Table 5.4. Nevertheless, list generation is time consuming, but, since the lists may be generated off-line and stored, this does not adversely impact the performance of the algorithm. Moreover, if new customers needed to be added to an extant set of lists, their list could simply be appended to the older set, so long as the problem parameters were consistent.

5.6.2 Single Instance Optimization

Initially, for both the Gold and Silver lines, customer migration lists were truncated to length at most four (we experimented with this later, see Section 5.6.5). This limitation made customers more selective and reduced solution times (typically to a few hours). The majority of this time was spent

	Customers	Typical Mean / Stdev of List Size	Typical Generation Time (hours)
Gold	15844	8.18 / 5.81	0.8
Silver	5278	606.18 / 191.86	28.5

Table 5.4: Customers, typical migration lists and generation times.

making incremental improvements or verifying optimality – in all instances the algorithm found an optimal or near-optimal solution (true gap within 0.5%) in under an hour.

As shown in Table 5.5, reducing product lines can lead to a significant increase in optimal expected profit. The column named "Profit Ratio" shows the ratio (profit from an optimal portfolio) / (profit from Deere's original portfolio). Table 5.5 also reports the number of configurations sold by Deere in the initial portfolio (Initial Configs)– this does *not* include those configurations produced but not sold. Our solutions reduce the number of configurations even below *these* levels, but our Service Level, defined as $(1 - \frac{\text{number of customers lost}}{\text{total number of customers}})$, remains high: 96% on the Gold line and virtually 100% on the Silver. Moreover, if the migration lists of customers who defect from the system are extended to their un-truncated length *without re-optimization*, the actual service levels (and profits) would increase. Note though that the solutions in Table 5.5 are to single instances, and as such show the *potential* for improvement. Whether or not a single portfolio can be constructed to perform this well across different instances, what we call the optimal *generic* portfolio, is a different question. This will be explored in Section 5.6.4

	Profit Ratio	Service Level	Initial Configs	Optimal Configs	Time (hours)
Gold	1.1853	96.30	698	282	1.62
Silver	1.0816	99.96	816	539	2.18

Table 5.5: Optimal solutions for the Gold and Silver lines.

An important measure of customer satisfaction is the percentage of customers who found their "favorite" product, i.e. who purchased machines at the top of their migration list. Table 5.6 shows that in both instances, at least 76% of the customers had either their first or second choice as part of the product line, and well over 90% had one of their top three. Customers are more focused around their top choices for the Gold line than for the Silver; this is a manifestation of the greater customer heterogeneity in the Silver line, we will see it repeated in later experiments.

	% Customers Buying			
	1st	2nd	3rd	4th
Gold	43.64	40.32	9.54	2.78
Silver	31.14	44.69	15.57	8.56

Table 5.6: Percentage of sales per position in migration list.

5.6.3 Replicate Experiments

Recognizing that the solution to a single instance would typically not be sufficient to construct an effective generic portfolio, we solve replicate instances with identical parameter settings, to identify

common characteristics of optimal portfolios. Using these results, we will explore different ways of building generic portfolios, in Section 5.6.4.

Table 5.7 reports the solution of ten instances for the Gold line. These ten instances' optimal configurations vary by less than ten percent, between 274 and 295, and the ratios of optimal profit to Deere's solution are even closer, differing by less than 0.01. Likewise, running times were in general similar, with one outlier. (This was instance 2, which also has the lowest profit and largest number of configurations, likely due to lists that were unusually disjoint.) Overall, this table paints a relatively consistent *high-level* picture of the optimal product line, particularly with respect to breadth and profit. Looking at the ten solutions in detail though, yields a slightly different picture.

	Profit	Service level %	Opt	Time	Choice %			
Instance	Ratio	trunc 4 / non trunc	Configs	(hours)	1st	2nd	3rd	4th
1	1.1853	96.30 / 97.08	282	1.62	43.64	40.32	9.54	2.78
2	1.1790	96.01 / 97.07	295	7.45	41.85	39.16	12.19	2.81
3	1.1866	96.52 / 97.13	283	0.63	43.45	41.23	9.11	2.73
4	1.1848	96.50 / 97.16	283	0.74	42.90	41.17	10.10	2.32
5	1.1855	96.55 / 97.26	291	1.05	45.32	37.57	10.91	2.76
6	1.1848	96.28 / 97.08	287	0.83	45.13	34.39	12.96	3.79
7	1.1847	96.23 / 97.00	279	0.94	42.79	40.63	9.81	2.99
8	1.1855	96.50 / 97.30	288	0.71	45.36	36.35	12.08	2.71
9	1.1871	96.01 / 96.74	280	0.84	43.81	37.74	11.05	3.41
10	1.1850	96.30 / 97.22	288	1.28	46.82	34.47	12.38	2.63

Table 5.7: Optimal solutions for ten instances of the Gold line.

In Figure 5.1 we display a histogram of the number of appearances of individual configurations in the ten optimal solutions, where for reasons of scale we show only the 612 configurations that appear at least once among these ten instances of the problem. From the figure we can identify the "core" of the optimal product line, the 103 configurations that appear in every optimal solution. On the other end of the spectrum, there are 245 configurations that appear only once or twice – these are configurations that fill out a portfolio, and likely are substitutes for each other. In between are the configurations that appear more frequently, but are not absolutely critical. We will see in Section 5.6.4 that this relatively large core (over 35% of any instance's optimal portfolio) implies that there are many different ways of constructing a good generic portfolio for the Gold line – so long as Deere gets the core right they will do quite well. The Silver line presents a more challenging situation.

We show similar results for the Silver line in Table 5.8 and Figure 5.2. Looking first at Table 5.8, the Silver line solutions are again largely consistent; there is one outlier with respect to the optimal number of configurations – instance 1 has 539, well below the others, this time likely due to *less* disjoint migration lists – but the ratios of optimal profit are nearly identical, varying no more than .003. Running times did show some variation, but were reasonable in all cases.

Despite this consistency, the frequency histogram generated by the Silver line experiments is qualitatively different from that of the Gold line. As seen in Figure 5.2, again showing only the 2208 configurations that appear at least once, there is a much smaller "core" to the product line – 94 configurations, or roughly one sixth of an optimal product line, appear in at least *nine* of the optimal solutions. In contrast to this, 1042 configurations, almost twice a product line, appear only once. There are two factors driving this dispersion: The first is the heterogeneous Silver customer base, as described in Section 5.6. The second factor is the (virtually) 100% service level in all ten Silver instances; due to the larger margins for these machines, even customers whose preferences make them

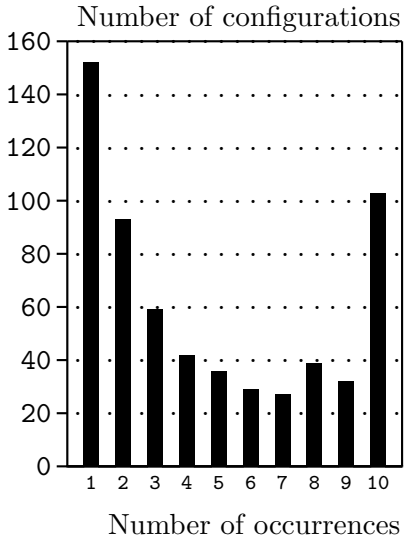

Figure 5.1: Histogram of Gold line configuration frequency in optimal solutions from Table 5.7.

	Profit	Service level %	Opt	Time	Choice %			
Instance	Ratio	trunc 4 / non trunc	Configs	(hours)	1st	2nd	3rd	4th
1	1.0816	99.96 / 99.96	539	2.18	31.14	44.69	15.57	8.56
2	1.0800	100 / 100	580	1.39	29.30	43.72	17.60	9.38
3	1.0817	100 / 100	570	1.04	29.12	44.00	18.39	8.49
4	1.0827	100 / 100	563	0.79	29.30	44.67	17.08	8.94
5	1.0803	100 / 100	582	0.82	29.46	46.51	15.86	8.18
6	1.0796	100 / 100	583	5.14	29.23	44.80	16.63	9.34
7	1.0800	100 / 100	575	1.01	30.44	43.80	16.52	9.24
8	1.0807	100 / 100	578	5.31	29.30	46.94	15.44	8.32
9	1.0820	100 / 100	582	0.90	26.86	45.05	17.96	10.13
10	1.0809	100 / 100	569	1.40	29.15	42.28	19.21	9.36

Table 5.8: Optimal solutions for ten instances of the Silver line.

outliers are being served. (By comparison, service levels are typically 96-97% on the Gold line.) Satisfying these outliers is expensive, but worthwhile, on the Silver line due to higher profit margins. *Thus high margins combine with customer heterogeneity to broaden product lines.*

This combination of a heterogeneous customer base and very high service level also increases the percentage of customers served with lower choices in their migration lists: the model forces more of them to accept substitute configurations. Whether such substitution is likely within a customer base, and what strategies may make such substitution more likely, is a question Deere (or any similar company) must study very closely. A salient question thus becomes whether *our algorithm* can ascertain whether such substitution is likely. We will see this in the next section.

5.6.4 Constructing an Optimal Portfolio

Based on the results of the previous section, in particular the relatively large core of the Gold line as compared to the Silver, we would expect constructing a good generic Gold portfolio to be easier than constructing one for the Silver line. To explore this, we tested two methods of constructing generic portfolios. In the first we simply tested four of the optimal portfolios for the Gold and Silver lines on

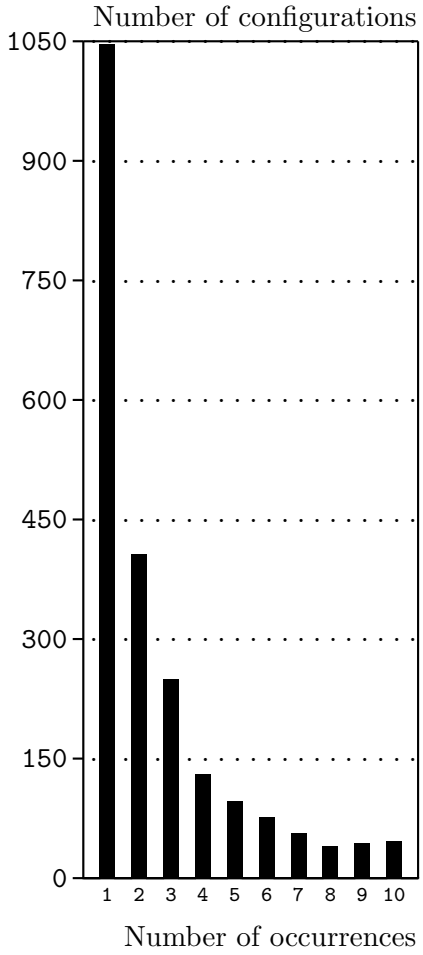

Figure 5.2: Histogram of Silver line configuration frequency in optimal solutions from Table 5.8.

ten additional randomly generated instances. The performance of this method is reported in the first two rows of Table 5.9.

		Profit	Service level %	Opt	Choice %			
Method	Line	Ratio	trunc 4 / non trunc	Configs	1st	2nd	3rd	4th
Single Instance	Gold	1.1499	93.82 / 95.43	289	44.16	35.54	9.11	3.00
Single Instance	Silver	0.9345	87.96 / 99.84	571	28.27	40.92	12.45	6.31
Union	Gold	1.1284	98.36 / 98.66	612	60.22	33.5	4.08	0.56
Union	Silver	.9922	100 / 100	2208	53.93	40.21	4.88	1.09

Table 5.9: Generic Portfolio Characteristics.

We see from the table that this provides good performance for the Gold line, but the Silver Line is not broad enough; its service level has declined dramatically. Note though that the performance for the Silver line rebounds when we use non-truncated lists – (virtually) all of the customers do want machines in the generic portfolios, but these machines are not among their top four choices. This is a manifestation of the fact that Silver customers place similar weights on different features, so our perturbation of features' importances can dramatically reshuffle lists. This implies that Deere must be careful when trying to reduce the Silver line – having features that customers value roughly

equivalently means that customers may be willing to substitute, but customer heterogeneity implies that there may be a significant amount of substitution, and this may come at a price. We will see how Deere acted on this insight in Section 5.7.

Given that a single optimal Silver portfolio may be too narrow, we tried the more conservative solution of taking every configuration that appeared in one of the optimal solutions, or the union of the elements on the histograms of Figures 5.1 and 5.2. As seen in the last two rows of Table 5.9, this is still quite good for the Gold line, although less than using a single instance, and better (but still not satisfactory) for the Silver, at least for the truncated lists. This reinforces the message that constructing a generic portfolio for the Gold line is relatively easy, due to this line's large core, but for the Silver it is more difficult, due to customer heterogeneity and high optimal service levels.

From Table 5.9, as well as the results in Section 5.6.3, it appears that for the Gold line a good generic portfolio can be constructed with approximately 300 configurations. The number of configurations in a good generic Silver line is less clear. To try to establish this, we used weighted sampling from Table 5.2 to construct four different generic Silver portfolios, and then varied each portfolio's size. We "offered" these portfolios to a single instance of the Silver line problem, plotting the profit of each of portfolio as a function of its size. Figure 5.3 shows the results of this experiment; a good target portfolio size for the Silver line appears to be around 1400 configurations, and 1200, twice a single optimal portfolio, appears to be a minimum.

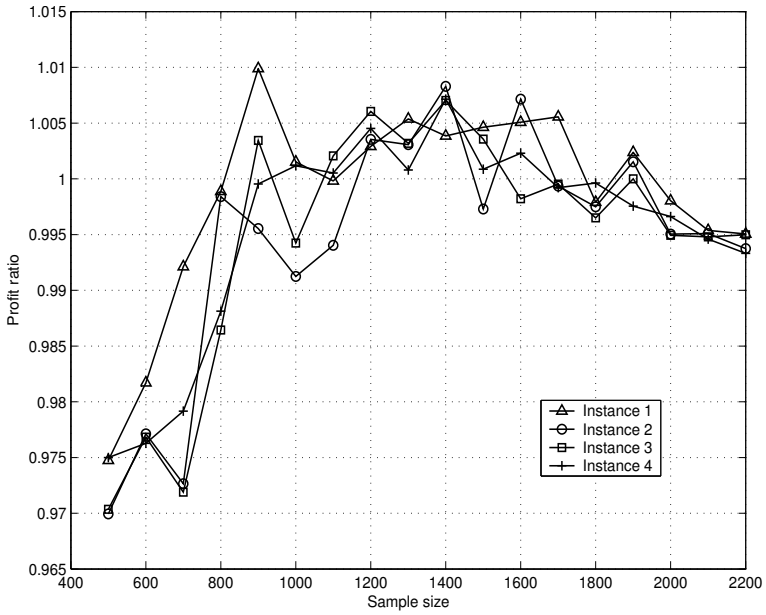

Figure 5.3: Profit versus portfolio size for four Silver Instances.

5.6.5 Sensitivity Analysis

In this section we conduct experiments varying parameters affecting: (i) customers' potential configurations, or their search space, in Section 5.6.5 and (ii) how customers evaluate these potential configurations, in Section 5.6.5. In the former case, we vary c, the number of features a customer may vary around C_0; the truncation point of the migration lists (default was four); and u and r, which affect reservation prices and utilities. With respect to how customers evaluate configurations, we experimented with changing n, the number of features having their relative importance scaled; the

magnitude of these variations; and finally β, which captures how true customers are to their purchases. All experiments were conducted on instances of the Gold line from Table 5.7.

Parameters Affecting Search Space

We first studied increasing c from two (the default for the Gold line) up through seven. Surprisingly, the effects of these changes were negligible; simply allowing customers to change more features from C_0 left migration lists, and hence optimal portfolios, largely unaffected. We hypothesize that this is because changing c did *not* affect how most customers evaluated configurations; the same set of configurations tended to be preferred, even though customers had more potential choices.

Does this insensitivity carry over to changing parameters affecting reservation prices and utilities, r and u, since these also only affect the configurations a customer will consider? No. Even though relaxing these constraints (making r and u larger) does not change how customers evaluate configurations, this *does* lead to product lines that have higher profits and contain fewer variants, by enabling our algorithm to concentrate customers around higher end models. (The correlation between utility and margin for the Gold line is .9038.) We see this in detail in Table 5.10, where we vary p and u for Gold line instances 1, 2, 3, and 10 from Table 5.7.

	Profit	Service level %	Opt	Time	Choice %			
Change	Ratio	trunc 4 / non trunc	Configs	(hours)	1st	2nd	3rd	4th
$r/2$	1.1295	96.79 / 97.22	296	0.51	61.03	24.38	8.71	2.67
$r/2$	1.1285	96.93 / 97.34	311	0.79	56.14	30.12	8.19	2.48
$r/2$	1.1279	96.52 / 97.15	276	1.14	64.37	21.62	7.88	2.64
$r/2$	1.1278	96.46 / 96.92	288	0.71	60.02	26.72	7.09	2.63
$2r$	1.2897	95.83 / 96.94	276	0.73	34.85	43.81	13.92	3.26
$2r$	1.2944	96.12 / 96.98	263	0.48	35.50	45.34	12.41	2.88
$2r$	1.2872	96.25 / 97.30	263	0.60	38.18	42.34	13.02	2.70
$2r$	1.2876	96.27 / 97.34	268	3.98	39.37	41.63	11.94	3.33
$u/2$	1.1670	96.50 / 97.21	298	0.44	55.54	30.08	8.66	2.23
$u/2$	1.1664	96.30 / 97.15	295	0.80	50.44	35.43	8.39	2.02
$u/2$	1.1661	96.35 / 97.10	286	1.32	54.82	31.27	8.12	2.16
$u/2$	1.1665	96.45 / 97.02	293	1.05	52.71	33.61	7.95	2.19
$2u$	1.2252	97.55 / 98.64	280	> 24 (0.14%)	27.03	46.01	19.32	5.19
$2u$	1.2270	97.07 / 98.45	260	> 24 (0.12%)	27.28	46.06	18.82	4.90
$2u$	1.2247	97.39 / 98.62	277	> 24 (0.18%)	26.62	48.06	17.51	5.21
$2u$	1.2244	97.15 / 98.50	268	> 24 (0.24%)	26.29	45.76	19.55	5.55

Table 5.10: Sensitivity analysis with respect to reservation price and utility for the Gold line.

Note that there is one less obvious effect of this change – the proportion of customers purchasing their first choice decreases dramatically as reservation price (or utility) relaxes – customers are being "steered" toward higher end products. Increasing u also makes the problem correspondingly more difficult to solve – for the four instances with $2u$ the algorithm only comes to within the optimality gap shown in parentheses in 24 hours. We saw very similar effects when increasing the truncation point for migration lists: Solution times and profits increased as service levels remained virtually unchanged, as optimal portfolios served the same number of customers using fewer configurations.

Summarizing the effects of changing c, r, u and truncation points, we conclude that simply giving customers more choices will not necessarily increase profits. *But,* if increasing their choices makes

customers more flexible, (or if such flexibility can be induced, see Section 5.7), then lines may be condensed and profits increased.

Parameters Affecting Evaluation

Now we alter how customers *evaluate* configurations, examining the effects of changing n, (number of features selected to have their importance rescaled); feature importance variation; and β.

Similar to c, changing n had no significant effect on our solutions. This is not overly surprising – recall from Section 5.4.2 that the features not selected also have their importances scaled, to keep the summed mean feature values constant. Thus all utilities are in fact being scaled for $n > 0$.

We next investigated changing the variance of feature importance, experimenting with Gold line instances 1, 2, 3, and 10 from Table 5.7. We found the Gold line to be very stable; Table 5.11 reports the results of optimizing these instances after increasing these variations by *twenty times*. At this point profits decrease by just over ten percent (essentially to Silver line levels) as service levels remain constant but require more configurations to be achieved. Increasing feature variability elevates customer heterogeneity, making the Gold line customers behave more like the Silver line's. Note though that the Gold line's customers still have more in common than their Silver counterparts: They agree on their first choices much more, as over eighty percent now receive their first choice. This "agreement" does not lead to higher profits as compared to Silver though, as Silver customers pay higher margins on average.

	Profit	Service level %	Opt	Time	Choice %			
Instance	Ratio	trunc 4 / non trunc	Configs	(hours)	1st	2nd	3rd	4th
1	1.0779	96.28 / 97.98	314	9.36	80.59	5.69	3.17	6.82
2	1.0781	96.51 / 98.14	332	3.74	80.42	6.27	3.53	6.29
3	1.0771	96.12 / 97.95	308	3.27	80.54	5.86	3.31	6.41
10	1.0773	96.33 / 98.16	326	3.27	80.69	5.66	3.70	6.29

Table 5.11: Optimal solutions of four Gold line instances when relative importances vary widely.

Finally we turn to the effects of changing β, how likely it is that C_0 is forced to the top of a migration list. In Figure 5.4 we see how the profit ratio, service level, and first choice probability change with β for Gold line instance 1 from Table 5.7. As β changes from zero to one (customers' preference for their purchased machine grows) profits decrease by approximately five percent, service levels drop slightly, and first choice percentage drops dramatically. (Although not shown, changes in the the breadth of the product line show no discernible pattern.) This effect of β on profit is somewhat surprising, as changing β does *not* change the contents of any customer's migration list, only the order in which the items appear: Any solution to an instance with $\beta = 0$ will satisfy *exactly* the same set of customers as in the $\beta = 1$ instance. So what is driving this change? The answer lies, again, in the correlation between utility and margin for the Gold line.

When $\beta = 0$ customers are sorting their choices solely by utility, so their first choices tend to be machines with large margins. The optimization offers these, giving customers their first choices while also reaping greater profits. Conversely, when $\beta = 1$, customers will choose C_0 whenever it is available. Thus to steer customers to higher margin machines the optimization does not offer as many first choices. Thus estimating β is critical, it affects the structure of the optimal product line and also has potentially significant ramifications regarding the number of customers who find their "favorite" machine.

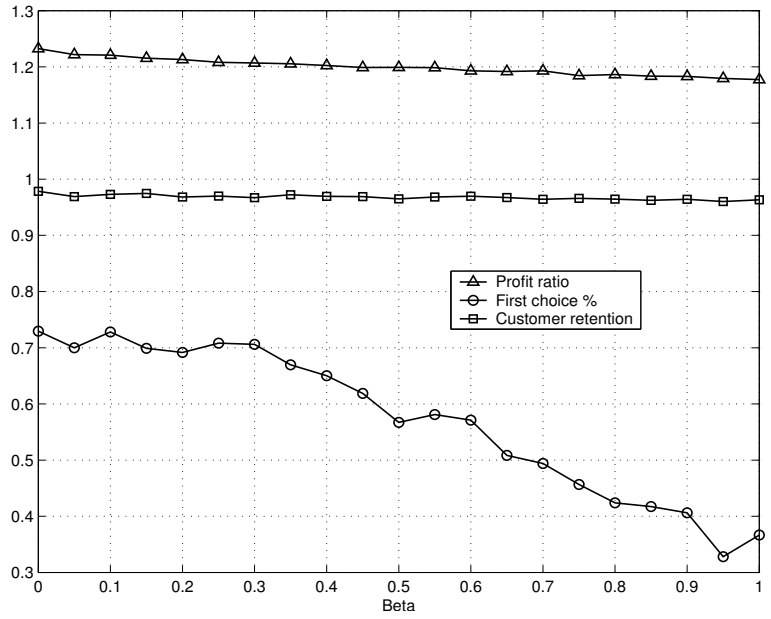

Figure 5.4: Profit, service level and first choice versus β for a single Gold line.

5.6.6 Algorithmic Modifications and Additional Constraints

We now briefly examine the effect on algorithm performance of (i) modifying the objective function to a step-wise form (as described in Section 5.5.3); and (ii) including additional managerial constraints in the MIP. Specifically, the constraints we add establish:

1. A lower bound $\mathcal{L}$ on configurations (to preserve product line breadth); $\sum_{j \in T} y_j \geq \mathcal{L}$;
2. An upper bound on configurations, $\mathcal{U}$ (to contain complexity costs); $\sum_{j \in T} y_j \leq \mathcal{U}$; or
3. A lower bound on service level, γ; $\sum_{i \in C} \sum_{j \in L_i} x_{ij} \geq \gamma$.

These are only a few of the possible modifications of the MIP; many others can be considered, for example ensuring that certain "flagship" or "core" configurations are present in the solution.

Adding constraints on the number of configurations (≥ 400 or ≤ 200) tends to speed the algorithm, while requiring a higher service level or using the stepwise complexity function tends to increase solution times, but they remained reasonable. Thus our algorithm, at least in these preliminary tests, is robust with respect to running time for typical problem variants.

Managerially, we examine how increasing the Gold line service level affects solutions. In Table 5.12 we report average profit and portfolio size for different service levels, over instances 1, 2, 3, and 10 from Table 5.7. As expected, increasing service level broadens the product line appreciably, but it has only a *slight* effect on profits: The extra cost of the additional configurations is almost compensated for by the additional revenue. Thus for the Gold line Deere need not be overly aggressive in reducing the line – keeping a broader line and serving more customers is still very profitable.

5.6.7 Summary of Computational Results

Our computational results illustrate a number of characteristics of Deere's problem, and portfolio optimization in general. With respect to Deere, the Gold and Silver lines exhibit significant differences

Service Level	Profit Ratio	Opt Configs
97%	1.1835	312
98%	1.1808	359
99%	1.1743	433

Table 5.12: Impact of service level constraints for Gold line.

when optimization is applied: The Gold line more readily accepts optimization, because the Gold line *customers* more readily accept optimization – they have more in common than their Silver line counterparts. As such a well-defined "core" of an optimal Gold product line is apparent in our replicate experiments. Any reduced line that includes this core should lead to a significant increase in profits, through offering fewer configurations and possibly "steering" customers to higher margin machines.

The Silver line is more tricky – there is a much smaller core visible through replication. We can trace this to a specific type of heterogeneity among Silver line customers – many Silver line configurations have roughly equal mean parts worth utility, so different customers may have quite different tastes, even though they are shopping for "comparable" machines (having similar total utility and price). When this fact is combined with the high Silver line margins, which imply that serving all customers is optimal, broader Silver lines result. For Deere this means that they should be more conservative in their efforts to reshape the Silver product line.

Our sensitivity analysis focused on those parameters that affect customer behavior. We isolate two different types of parameters affecting customer behavior: Those that affect the configurations a customer will consider, and those that affect how these configurations are evaluated. In general, widening the search space without changing customers' evaluations does not change the problem. In contrast, changing how customers think, by relaxing their reservation price or utilities, making them more or less variable in their evaluation of feature values, or tying them more or less closely to their purchased configuration does change the problem considerably. In a nutshell, if customers can be made more flexible lines can more easily be consolidated, and profits can be improved. This is the message we brought to Deere, and this is the message that shaped their implementation.

5.7 Implementation at Deere

Given the insights from our analysis, Deere decided that rather than remove configurations, they would offer discounts on specific options to "steer" customers toward a smaller set of configurations. This reduced set may be thought of as the "core" of the product line; we will call this set C. Clearly offering greater incentives will increase migration, but also increase costs; thus there is a trade-off that regulates how much effort Deere should exert to induce those customers outside of C to migrate into C. The results of our analysis helped Deere determine specific migration targets for each of the two product lines (both in excess of 50%) for the percentage of customers who migrate, as well as the level of incentives to offer.

Based on the demand before and after the incentives were implemented, Deere has already achieved their targets, by providing discounts of 10-15% on selected options (accounting for 0.5% to 2% of total model cost) without having to publicly announce a product line compression, or "force" customers away from their first choices. Deere also found, in line with our results, that more migration is possible, with smaller incentives, in the Gold line than in the Silver.

This implementation raises a question: Without actually discontinuing products, can Deere reap

profits from more concentrated product lines, especially in a primarily make-to-stock environment? The short answer is yes, so far in the tens of millions of dollars. But where are these profits coming from? Previously, Deere built more configurations, some of which were not even sold once. Now Deere builds fewer variants in advance (but still provides customers with any feasible configuration, custom built, that they want). Moreover, a smaller set of configurations are eventually sold, reducing the complexity of servicing these configurations throughout their lifetime.

5.8 Summary and Conclusions

While the literature contains much work on optimizing product lines, there have been no industrial applications, nor even any algorithm presented, capable of solving problems of the size and level of complexity found at Deere. We optimally solve such instances – with tens of thousands of customers and possible configurations – in several hours. One factor critical to this efficiency is our use of customer migration lists, capturing customer behavior in a form amenable to optimization.

Leveraging the efficiency of our algorithm, we solve different variants of Deere's basic model, as well as multiple randomized replications of the same instance. This helps us to identify the configurations that form the core of any good solution, useful in building optimal "generic" portfolios. It also led to several insights into the product line optimization problem, specifically regarding the effects of different modeling choices for customer flexibility and heterogeneity. Moreover, by quantifying the benefits of customer flexibility, we have helped Deere design and incentive program which has proved to be very effective at steering customers to a smaller set of configurations, elevating profits by tens of millions of dollars. In addition, Deere's incentive program has helped them avoid the appearance of denying customers their first choice, by discontinuing models.

Our work is being applied to other product lines at Deere, and can be to other companies as well. This holds the promise of greater efficiencies, leading to increased profits as well as greater customer satisfaction, as resources are focused on products that reflect their desires.

Bibliography

J. Abeggeln and G. Stalk. *KAISHA, The Japanese Corporation*. Basic Books, 1985.

T. Achterberg. Scip: A framework to integrate constraint and mixed integer programming. Technical Report 04-19, Konrad-Zuse-Zentrum für Informationstechnik (ZIB), Berlin, 2004.

F. Ajili and M. Wallace. Hybrid problem solving in eclipse. In M. Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*, pages 169–201. Kluwer, 2003.

S. Anderson. Measuring the impact of product mix heterogeneity on manufacturing overhead cost. *The Accounting Review*, 70(3):363–387, 1995.

I. D. Aron, J. N. Hooker, and T. H. Yunes. SIMPL: A system for integrating optimization techniques. In J.-C. Régin and M. Rueher, editors, *Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3011 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 2004.

E. Balas. Disjunctive programming: Properties of the convex hull of feasible points. *Discrete Applied Mathematics*, 89:3–44, 1998.

P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling*. Kluwer, 2001.

C. Beck and P. Refalo. A hybrid approach to scheduling with earliness and tardiness costs. *Annals of Operations Research*, 118:49–71, 2003.

J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

T. Benoist, E. Gaudin, and B. Rottembourg. Constraint programming contribution to benders decomposition: A case study. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *Lecture Notes in Computer Science*, pages 603–617. Springer-Verlag, 2002.

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*. Elsevier Science, 1995.

M. Berkelaar. The LP_SOLVE linear programming solver. ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.

G. Blau, B. Mehta, S. Bose, J. Pekny, G. Sinclair, K. Keunker, and P. Bunch. Risk management in the development of new products in highly regulated industries. *Computers and Chemical Engineering*, 24:659–664, 2000.

A. Bockmayr and F. Eisenbrand. Combining logic and optimization in cutting plane theory. In H. Kirchner and C. Ringeissen, editors, *Proceedings of the Third International Workshop on Frontiers of Combining Systems (FroCos)*, volume 1794 of *Lecture Notes in Artificial Intelligence*, pages 1–17. Springer-Verlag, March 2000.

A. Bockmayr and T. Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.

S. Bollapragada, O. Ghattas, and J. N. Hooker. Optimal design of truss structures by mixed logical and linear programming. *Operations Research*, 49(1):42–51, 2001.

S. Broniarczyk, W. Hoyer, and L. McAlister. Consumers' perceptions of the assortment offered in a grocery category: The impact of item reduction. *Journal of Marketing Research*, 35:166–176, 1998.

A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. Ecl^{i}ps^e: An introduction. Technical Report 03-1, IC-Parc, Imperial College, London, 2003.

F. Chen, J. Elishberg, and P. Zipkin. Customer preferences, supply-chain costs, and product-line design. *Research Advances in Product Variety Management*, pages 123–144, 1998.

K. Chen and W. Hausman. Technical note: Mathematical properties of the optimal product line selection problem using choice-based conjoint analysys. *Management Science*, 46(2):327–332, 2000.

J. K. Chong, T. Ho, and C. Tang. Product structure, brand width, and brand share. *Research Advances in Product Variety Management*, pages 39–64, 1998.

Y. Colombani and S. Heipcke. Mosel: An extensible environment for modeling and programming solutions. In *Proceedings of the International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, 2002.

Y. Colombani and S. Heipcke. Mosel: An overview. Dash Optimization white paper, 2004.

G. Cornuéjols, M. Karamanov, and Y. Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 2004. To appear.

G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

Dash Optimization, Inc. *XPRESS-MP: User Manual*, 2000. Version 12.11.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

X. De Groote. Flexibility and marketing/manufacturing coordination. *International Journal of Production Economics*, 36:153–167, 1994.

G. Dobson and S. Kalish. Positioning and pricing a product line. *Marketing Science*, 7(2):107–125, 1988.

G. Dobson and S. Kalish. Heuristics for pricing and positioning a product line using conjoint and cost data. *Management Science*, 39(2):160–175, 1993.

K. Easton, G. Nemhauser, and M. Trick. Solving the traveling tournament problem: A combined integer programming and constraint programming approach. In *Proceedings of the Fourth International Conference on the Practice and Theory of Automated Timetabling (PATAT)*, 2002.

A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, November 2001.

M. Fisher and C. Ittner. The impact of product variety on automobile assembly opertations: Empirical evidence and simulation analysis. *Management Science*, 45(6):771–786, 1999.

M. Fisher, A. Jain, and J. MacDuffie. Strategies for product variety: Lessons from the auto industry. *Redesigning the Firm*, pages 116–154, 1995.

F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 1999.

F. Focacci, A. Lodi, and M. Milano. Cutting planes in constraint programming: A hybrid approach. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1894 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 2000.

G. Foster and M. Gupta. Manufacturing overhead cost driver analysis. *Journal of Accounting and Economics*, 12:309–337, 1990.

F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

P. Green and A. Krieger. Models and heuristics for product line selection. *Marketing Science*, 4(1): 1–19, 1985.

M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.

G. Gutin and A. P. Punnen, editors. *Traveling Salesman Problem and Its Variations*. Kluwer, 2002.

M. T. Hajian, H. El-Sakkout, M. Wallace, J. M. Lever, and E. B. Richards. Toward a closer integration of finite domain propagation and simplex-based algorithms. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 1996.

R. Hayes and S. Wheelright. *Restoring our Competitive Edge*. John Wiley, 1984.

R. Henkoff. New management secrets from japan. *Fortune*, pages 135–146, November 1995.

S. Honkomp, G. Reklaitis, and J. Pekny. Robust planning and scheduling of process development projects under stochastic conditions. Presented at the AICHE annual meeting, Los Angeles, CA, 1997.

J. N. Hooker. Logic-based methods for optimization. In A. Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 336–349. Springer-Verlag, 1994.

J. N. Hooker. Inference duality as a basis for sensitivity analysis. In E. C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1118 of *Lecture Notes in Computer Science*, pages 224–236. Springer-Verlag, 1996.

J. N. Hooker. Constraint satisfaction methods for generating valid cuts. In D. L. Woodruff, editor, *Advances in Computational and Stochastic Optimization, Logic Programming and Heuristic Search*, pages 1–30. Kluwer, Dordrecht, 1997.

J. N. Hooker. Inference duality as a basis for sensitivity analysis. *Constraints*, 4:104–112, 1999.

J. N. Hooker. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 2000.

J. N. Hooker. Logic, optimization and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, 2002.

J. N. Hooker. A framework for integrating solution methods. In H. K. Bhargava and M. Ye, editors, *Computational Modeling and Problem Solving in the Networked World*, pages 3–30. Kluwer, 2003. Plenary talk at the Eighth INFORMS Computing Society Conference (ICS).

J. N. Hooker. A hybrid method for planning and scheduling. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *Lecture Notes in Computer Science*, pages 305–316. Springer-Verlag, 2004.

J. N. Hooker. A hybrid method for planning and scheduling, 2005a. To appear in *Constraints*.

J. N. Hooker. A search-infer-and-relax framework for integrating solution methods. Presented at the INFORMS National Meeting, San Francisco, CA, 2005b.

J. N. Hooker. A search-infer-and-relax framework for integrating solution methods. In R. Barták and M. Milano, editors, *Proceedings of the Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3709 of *Lecture Notes in Computer Science*, pages 314–327. Springer-Verlag, 2005c.

J. N. Hooker and M. A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96–97(1–3):395–442, 1999.

J. N. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96: 33–60, 2003.

J. N. Hooker, G. Ottosson, E. Thorsteinsson, and H.-J. Kim. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pages 136–141. MIT Press, 1999.

J. N. Hooker, G. Ottosson, E. S. Thorsteinsson, and K.-J. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review*, 15:11–30, 2000.

J. N. Hooker and H. Yan. Logic circuit verification by benders decomposition. In V. Saraswat and P.Ṽan Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 267–288. MIT Press, 1995.

J. N. Hooker and H. Yan. A relaxation for the cumulative constraint. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *Lecture Notes in Computer Science*, pages 686–690. Springer-Verlag, 2002.

ILOG S. A. The CPLEX mixed integer linear programming and barrier optimizer. On the web at http://www.ilog.com/products/cplex/.

V. Jain and I. Grossmann. Resource-constrained scheduling of tests in new product development. *Industrial and Engineering Chemistry Research*, 38(8):3013–3026, 1999.

V. Jain and I. E. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing*, 13(4):258–276, 2001.

U. Junker, S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A framework for constraint programming based column generation. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1713 of *Lecture Notes in Computer Science*, pages 261–274. Springer-Verlag, 1999.

B. Kahn. Consumer variety-seeking among goods and services. *Journal of Retailing and Consumer Services*, 2(3):139–148, 1995.

S. Kekre. Performance of a manufacturing cell with increased product mix. *IIE Transactions*, 19(3): 329–339, 1987.

S. Kekre and K. Srinivasan. Broader product line: A necessity to achieve success? *Management Science*, 36(10):1216–1231, 1990.

R. Kohli and R. Sukumar. Heuristics for product-line design using conjoint analysis. *Management Science*, 36(12):1464–1478, 1990.

E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 1985.

S. Leipert. VBCTOOL: A graphical interface for visualization of branch-and-cut algorithms. On the web http://www.informatik.uni-koeln.de/old-ls_juenger/projects/vbctool.html.

K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

R. McBride and F. Zufryden. An integer programming approach to the optimal product line selection problem. *Marketing Science*, 7(2):126–140, 1988.

M. Milano. *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer, 2003.

M. Milano, G. Ottosson, P. Refalo, and E. S. Thorsteinsson. The role of integer programming techniques in constraint programming's global constraints. *INFORMS Journal on Computing*, 14(4): 387–402, 2002.

L. Morgan, R. Daniels, and P. Kouvelis. Marketing/manufacturing trade-offs in product line management. *Management Science*, 33:949–962, 2001.

S. Nair, L. Thakur, and K.W. Wen. Near-optimal solutions for product line design and selection: Beam search heuristics. *Management Science*, 41(5):767–785, 1995.

G. L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society*, 43(5):443–457, 1992.

G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988.

N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

G. Ottosson, E. S. Thorsteinsson, and J. N. Hooker. Mixed global constraints and inference in hybrid CLP-IP solvers. In *CP'99 Post Conference Workshop on Large Scale Combinatorial Optimization and Constraints*, pages 57–78, 1999.

G. Ottosson, E. S. Thorsteinsson, and J. N. Hooker. Mixed global constraints and inference in hybrid CLP-IP solvers. *Annals of Mathematics and Artificial Intelligence*, 34:271–290, 2002.

M. W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5: 199–215, 1973.

J. Quelch and D. Kenny. Extend profits: Not product lines. *Harvard Business Review*, 72(5):153–160, 1994.

A. Raleigh. "Product Logic". New England Council for Operations Excellence Seminar: Managing Complexity for Competitive Advantage, June 2003.

N. Raman and D. Chhajed. Simultaneous determination of product attributes and prices, and production processes in product-line design. *Journal of Operations Management*, 12:187–204, 1995.

K. Ramdas. Managing product variety: An integrative review and research directions. *Production and Operations Managment*, 12(1):79–101, 2003.

T. Randall, K. Ulrich, and D. Reibstein. Brand equity and vertical product line extent. *Marketing Science*, 17(4):756–379, 1998.

R. Rasmussen and M. Trick. A benders approach for the minimum break scheduling problem. Presented at the INFORMS National Meeting, San Francisco, CA, 2005.

C. R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Wiley, 1993.

P. Refalo. Tight cooperation and its application in piecewise linear optimization. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1713 of *Lecture Notes in Computer Science*, pages 375–389. Springer-Verlag, 1999.

P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1894 of *Lecture Notes in Computer Science*, pages 369–383. Springer-Verlag, 2000.

R. Rodošek, M. Wallace, and M. T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operations Research*, 86:63–87, 1999.

J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 362–367, 1994.

N. V. Sahinidis and M. Tawarmalani. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*. Kluwer Academic Publishers, 2003.

Z. Schiller. Make it simple. *Business Week*, pages 96–104, September 1996.

C. Schmidt and I. Grossmann. Optimization models for the scheduling of testing tasks in new product development. *Industrial and Engineering Chemistry Research*, 35(10):3498–3510, 1996.

M. Sellmann and T. Fahle. Constraint programming based lagrangian relaxation for a multimedia application. In *Proceedings of the Third International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, 2001.

G. Stalk Jr. and A. Webber. Japan's dark side of time. *Harvard Business Review*, 71(4):93–102, 1993.

U. Thonemann and M. Brandeau. Optimal commonality in component design. *Operations Research*, 48(1):1–19, 2000.

E. S. Thorsteinsson. Branch-and-Check: A hybrid framework integrating mixed integer programming and constraint logic programming. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, November 2001.

E. S. Thorsteinsson and G. Ottosson. Linear relaxations and reduced-cost based propagation of continuous variable subscripts. *Annals of Operations Research*, 115:15–29, 2001.

C. Timpe. Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, 24:431–448, 2002.

E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

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.

P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

P. Van Hentenryck, I. Lustig, L. Michel, and J. F. Puget. *The OPL Optimization Programming Language*. MIT Press, 1999.

W. J. Van Hoeve. A hybrid constraint programming and semidefinite programming approach for the stable set problem. In F. Rossi, editor, *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *Lecture Notes in Computer Science*, pages 407–421. Springer-Verlag, 2003.

M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. *ICL Systems Journal*, 12:159–200, 1997.

H. P. Williams and H. Yan. Representations of the all_different predicate of constraint satisfaction in integer programming. *INFORMS Journal on Computing*, 13(2):96–103, 2001.

L. A. Wolsey. *Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 1998.

H. Yan and J. N. Hooker. Tight representations of logical constraints as cardinality rules. *Mathematical Programming*, 85:363–377, 1999.

C. Yano and G. Dobson. Product line design for diverse markets. *Global Supply Chaina nd Technology Management*, pages 152–158, 1998a.

C. Yano and G. Dobson. Profit-maximizing product line design, selection and pricing with manufacturing cost consideration. *Research Advances in Product Variety Management*, pages 103–122, 1998b.

T. H. Yunes. On the sum constraint: Relaxation and applications. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *Lecture Notes in Computer Science*, pages 80–92. Springer-Verlag, 2002.

T. H. Yunes, A. V. Moura, and C. C. de Souza. Exact solutions for real world crew scheduling problems. Presented at the INFORMS National Meeting, Philadelphia, PA, 1999.

T. H. Yunes, A. V. Moura, and C. C. de Souza. Hybrid column generation approaches for urban transit crew management problems. *Transportation Science*, 39(2):273–288, 2005.