Abstract of "*A unifying framework for modeling and solving optimization problems*" by Ionuț D. Aron, Ph.D., Brown University, May, 2007

The idea of a unifying framework for optimization has been around for about a decade, but very few attempts have been made at providing a description of the major components and their interaction in such a framework. A notable exception is the recent book by HOOKER[50], which builds on the earlier work of CHANDRU AND HOOKER[28], HOOKER[48] and ARON, HOOKER AND YUNES[6] to provide a consistent view of optimization techniques as they are used in mathematical and constraint logic programming. The framework we describe here has evolved from a joint work with HOOKER AND YUNES[6], and it provides the basic blocks for a modeling language of metaconstraints, as well as the mechanisms that a generic metasolver needs in order to solve a problem according to the specification given by the metaconstraints. The thesis on which this framework is based is that problem solving is a combination of two fundamental steps, *search* and *inference*, whose efficient utilization requires modeling and logical deduction across theory lines. Such deduction is made possible by equipping metaconstraints with reformulation rules for multiple theories and using these rules throughout the search to draw inference from each of those theories.

Abstract of "*A unifying framework for modeling and solving optimization problems*" by Ionuţ D. Aron, Ph.D., Brown University, May, 2007

The idea of a unifying framework for optimization has been around for about a decade, but very few attempts have been made at providing a description of the major components and their interaction in such a framework. A notable exception is the recent book by HOOKER[50], which builds on the earlier work of CHANDRU AND HOOKER[28], HOOKER[48] and ARON, HOOKER AND YUNES[6] to provide a consistent view of optimization techniques as they are used in mathematical and constraint logic programming. The framework we describe here has evolved from a joint work with HOOKER AND YUNES[6], and it provides the basic blocks for a modeling language of metaconstraints, as well as the mechanisms that a generic metasolver needs in order to solve a problem according to the specification given by the metaconstraints. The thesis on which this framework is based is that problem solving is a combination of two fundamental steps, *search* and *inference*, whose efficient utilization requires modeling and logical deduction across theory lines. Such deduction is made possible by equipping metaconstraints with reformulation rules for multiple theories and using these rules throughout the search to draw inference from each of those theories.

A unifying framework for modeling and solving optimization problems

by

Ionuț D. Aron

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May, 2007

© Copyright 2007 by Ionuț D. Aron

This dissertation by Ionuț D. Aron is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ____________________

____
Pascal Van Hentenryck, Director
Brown University,
Department of Computer Science

Recommended to the Graduate Council

Date ____________________

____
John N. Hooker, Reader
Carnegie Mellon University,
Tepper School of Business

Date ____________________

____
Meinolf Sellmann, Reader
Brown University,
Department of Computer Science

Approved by the Graduate Council

Date ____________________

____
Sheila Bonde
Dean of the Graduate School

Acknowledgments

I wish to thank the people whose advice, support and collaboration have made this work possible and enjoyable: my advisor, Pascal Van Hentenryck, and my committee members John Hooker, Egon Balas and Meinolf Sellmann. The theme of this work is a direct reflection of my collaboration with them.

I would also like to thank the various people with whom I have worked or interacted throughout this entire period, my fellow graduate students, faculty and staff at Brown's Computer Science Department and Carnegie Mellon's Tepper School of Business. They have provided a challenging and professional environment.

None of this work would have been possible without the unconditional and continuing moral support of my family and friends across the ocean. I have felt their presence throughout and it was this presence that kept me going. This work is dedicated to them.

Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Theories: language and logic . . . 2

1.2 Towards a unified method: previous work . . . 5

1.2.1 General purpose solvers . . . 6

1.2.2 Modeling languages . . . 6

1.3 Organization . . . 7

2 Inference via reformulation 9

2.1 The importance of reformulation . . . 9

2.2 A special case: relaxation . . . 12

3 Optimization theories 15

3.1 Integer and disjunctive programming . . . 15

3.1.1 Disjunctive linear constraints . . . 17

3.1.2 Inference via reformulation . . . 18

3.1.3 Lift and project as an inference technique . . . 27

3.2 Constraint programming . . . 37

3.2.1 Modeling and solution process . . . 37

4 A unifying framework **39**
4.1 Modeling . . . 40
4.1.1 Metaconstraints . . . 41
4.2 Solution process . . . 44
4.2.1 Search . . . 45
4.2.2 Inference . . . 49
4.3 Example: Production planning . . . 55
4.3.1 Modeling . . . 56
4.3.2 Solution process . . . 58
4.3.3 Experimental results . . . 60
4.4 Example: Product configuration . . . 62
4.4.1 Modeling . . . 62
4.4.2 Solution process . . . 63
4.4.3 Experimental results . . . 65
4.5 Example: Logic-based Benders decomposition . . . 66
4.5.1 Modeling . . . 66
4.5.2 Solution process . . . 67
4.5.3 Experimental results . . . 70

5 Applications **71**
5.1 The robust spanning tree problem . . . 71
5.1.1 Inference . . . 76
5.1.2 Search . . . 80
5.1.3 Experimental results . . . 81

Bibliography **86**

A **94**
A.1 Lift and project cut generator: algorithm details . . . 94

⋆ Parts of this work have been published in UNCERTAINTY IN ARTIFICIAL INTELLIGENCE[4], INTEGRATION OF AI AND OR IN CP[6] and OPERATIONS RESEARCH LETTERS[5].

List of Tables

4.1 Production planning: search nodes and CPU time. 61
4.2 Product configuration: search nodes and CPU time. 65
4.3 Parallel machine scheduling: long processing times. 70
4.4 Parallel machine scheduling: short processing times. 70

5.1 Robust spanning tree: notations 76
5.2 Robust spanning tree: average CPU time 83

A.1 Lift and project: properties of the problems in MIPLIB 2003 109
A.2 Lift and project: properties of non-modularized cuts 110
A.3 Lift and project: gap closed by non-modularized cuts 111
A.4 Lift and project: properties of modularized cuts 112
A.5 Lift and project: gap closed by modularized cuts 113

List of Figures

3.1 Generating lift-and-project cuts in the lower dimensional space 31

4.1 Modeling: example of a metaconstraint with reformulation rules 43
4.2 Modeling: metaconstraint with multiple reformulation rules 43
4.3 General algorithm to solve an optimization problem P 44
4.4 Modeling: example of a metaconstraint with search rules 47
4.5 Modeling: example of a metaconstraint enforcing integrality of variables . 47
4.6 Modeling: example of search module specification 48
4.7 Search algorithm based on metaconstraint search rules 49
4.8 Modeling: alldifferent with reformulation in multiple languages 51
4.9 Modeling: example of global inference specification 53
4.10 Inference algorithm based on metaconstraint inference rules 54
4.11 Production planning: a semicontinuous piecewise linear function $f_i(x_i)$. . 55
4.12 Production planning: integrated model 58
4.13 Production planning: convex hull relaxation (shaded area) of $f_i(x_i)$. . . . 60

5.1 Robust spanning tree: robust deviation 73
5.2 Robust spanning tree: the search algorithm 75
5.3 Robust spanning tree: finding the largest cost of an edge 79
5.4 Robust spanning tree: a class 7 network 81
5.5 Robust spanning tree: a class 8 network 82
5.6 Robust spanning tree: results using CP on class 1. 84
5.7 Robust spanning tree: results using CP on class 7. 84
5.8 Robust spanning tree: results using CP on class 8. 85

A.1 Algorithm for generating cuts via pivoting 95
A.2 Algorithm for generating lift-and-project cuts 96
A.3 Finding a pivoting row (i.e. variable to be removed from $\tilde{\mathcal{B}}$) 97
A.4 Finding the best pivoting column in the current basis $\tilde{\mathcal{B}}$ 98
A.5 Changing the current basis $\tilde{\mathcal{B}}$ in the LP tableau 98
A.6 Computing the CGLP objective given by a prospective pivot (γ) 99
A.7 Computing coefficients for intersection cuts 100
A.8 Computing strengthened coefficients for intersection cuts 100
A.9 Computing the coefficient of a row after a prospective pivot (γ) 100
A.10 Pivoting in the $\tilde{\mathcal{B}}$ tableau 101
A.11 Computing row coefficients resulting from disjunctive modularization . . 102
A.12 Compute CGLP reduced cost 102
A.13 Determining the type of a variable 103
A.14 Computing rhs for intersection cuts 103
A.15 Intersection cut 103
A.16 Disjunctive modularization (integrality based strengthening) 103
A.17 Mixed integer Gomory cut 104
A.18 Cut strengthening using integrality conditions 104
A.19 Perturb a row of the $\tilde{\mathcal{B}}$ tableau 104
A.20 Remove perturbation from a row of the $\tilde{\mathcal{B}}$ tableau 104
A.21 Retrieving the tableau from XPRESS 105
A.22 Function which returns the current bound of variable x_j 106
A.23 Adjusting a row to reflect that all nonbasics are at a bound of zero . . . 106
A.24 Computing the right hand side from XPRESS 107

A unifying framework for modeling and solving optimization problems

Ionuț D. Aron

Brown University

May, 2007

Chapter 1

Introduction

Whenever we are faced with a new problem, we attempt to identify existing theories, or perhaps create new ones that would provide us with (1) a language to express the problem and (2) axioms, inference rules and theorems to reason about it. There may be more than one suitable theory, so part of the problem solving process is to also decide which one is best equipped (perhaps has more advanced theorems, and therefore might require the smallest number of extra inference steps) to solve the problem at hand as efficiently and conveniently as possible. For instance, there are problems to which a geometric reasoning is best, and thus we have *geometry*, with a language, axioms and theorems as a tool for solving such problems. Other problems are better suited for an algebraic reasoning, and for them we use *algebra*. There are problems where *set theory* is best, and so on. It is also not uncommon to see elements of two or more theories being used in solving a problem. This is particularly true for optimization problems. Even though we have specific theories for certain classes of problems, such as *linear programming*, *pure and mixed integer linear programming*, *nonlinear programming* and *constraint programming*, the literature is rich in examples in which elements of these theories are combined in various ways in order to obtain solutions to difficult problems. The particular combination depends on the problem and even though it could be done in arbitrary ways, certain patterns are used more frequently than others.

A unifying framework

The scope of this thesis is to provide a unifying framework under which these common patterns can be employed in a systematic manner in order to solve computational optimization problems. This framework is based on the following thesis:

> *The problem solving process, in optimization in particular, is a combination of two fundamental steps,* search *and* logical inference*, whose efficient utilization requires modeling and deduction across theory lines.*

The search step is used to advance towards a proof (solution) and it generally involves making uninformed decisions. Logical inference, or simply *inference*, plays a dual role, by detecting infeasible search directions and, when possible, preventing the search from even trying directions which can be proved wrong a priori.

1.1 Theories: language and logic

A scientific theory is a system of sentences which are accepted as true and which may be called *laws* or *statements*. In mathematics, these statements follow one another in a definite order according to certain principles (syntax) and are accompanied by considerations intended to establish their validity (semantics). More precisely, a theory consists of a linguistical part (*language*) and a logical part (*logic*). The language includes elementary *symbols* (constants, variables and logical connectives), as well as more complex constructions such as *terms*, *formulas* and *sentences*, all described by a grammar (*syntax*) and having a well defined meaning (*semantics*). The logical part allows us to manipulate elements of the language in order to establish new laws. It consists of *logical axioms* and *inference rules*. In order to *solve* a given problem with a particular theory, we must first state it using the linguistical elements of the theory and then apply the logical elements to construct a proof that the formed statement is either true or false. In the context of optimization, the language provides the tools we need to create a *model* of the problem, while the logic gives us the tools to use during the solution process.

Searching for a proof: variable elimination

Among the symbols occurring in mathematical models (as well as in theorems and proofs) we distinguish *constants* and *variables*. For instance, in arithmetic, which is the theory concerned with the investigation of the general properties of numbers, relations between numbers and operations on numbers, we encounter such constants as "number", "zero"/"0", "one"/"1", "sum"/"+" and so on. Constants have a well determined meaning, which remains unchanged throughout the course of proofs. Variables, on the other hand, do not posses any meaning by themselves. Thus, while questions like *"Is zero an even number?"* can always be answered in the affirmative or negative (the answer may be true or false, but at any rate it is meaningful), a question concerning a variable "x", for example the question *"Is* **x** *an even number?"*, cannot be answered meaningfully. So even though they are syntactically correct with respect to the linguistical part, constructions like "**x** *is an even number"* do not actually express a definite assertion and can neither be confirmed or refuted. We only obtain a sentence whose truth value can be established within the theory if we somehow manage to remove the variable "x" from the statement and replace it with symbols whose meaning is well defined.

One way in which we can achieve this is to directly replace "x" with a constant denoting a definite number; thus, for instance, if "x" is replaced by the symbol "1", the result is a false statement whereas a true statement arises on replacing "x" by "2" or "10". A more desirable (but not always possible) way is to apply symbolic manipulation to the statement in order to obtain an equivalent one without variables. For example, in order to decide whether the statement "$\exists x \in \mathrm{R} : ax^2 + bx + c = 0$" is true, we know that it is sufficient to examine the equivalent statement "$b^2 - 4ac \geq 0$", which can be obtained from the original through symbolic manipulation (elimination of variables, or, more precisely, of quantifiers). For simple problems, like the one above, this type of variable elimination (through symbolic manipulation) is possible, but for more complex problems, we need to rely on the first method, that of directly replacing variables with symbols. This method is known in the optimization and artificial intelligence communities as **search**. It typically involves two decisions: which variable to eliminate at a given stage,

and what constant to replace it with.[1] Since there are no general rules for either of these decisions that would guarantee a short proof, search needs to rely on guidance from the logical part of the theory at hand. This guidance comes in the form of **inference** and allows the search to avoid looking at all possible variable-constant substitutions.

Using inference to guide the search

In a given theory, logical axioms are universally true formulas and inference rules are elementary steps of the logical reasoning. For example, in propositional calculus, logical reasoning is driven by rules like *modus ponens* ($\text{"from } A \rightarrow B \text{ and } A, \text{ infer } B\text{"}$) and *modus tollens* ($\text{"from } A \rightarrow B \text{ and not } B, \text{ infer not } A\text{"}$). A theorem is either an axiom or an immediate consequence of other theorems (by application of some inference rule). A proof is a sequence of theorems such that each theorem is either and axiom or an immediate consequence of the theorems before it. The process through which we find the *immediate* consequences of the existing theorems is called **inference**.

Inference assists the solution process in two important, distinct ways. One one hand, the search should be prevented from attempting directions which can be proved *a priori* to lead to false statements. When such a priori proofs of infeasibility are not possible, the inference step is called upon after the search picks a direction and commits to it, since the effects of that decision may be easier to evaluate *a posteriori*. This time, inference plays a corrective role: it checks whether the most recent decision has created a false statement and if so it informs the search component that the decision must be undone.

Inference from multiple theories

While the choice of a language to express the problem at hand is very important, there is no reason why the inference step should be based on a single theory. Indeed, if we manage to properly express the current state of the proof (search) in multiple languages, we could use each of the corresponding theories in order to decide on the validity of the proof so far and on the best way to proceed.

[1] For some problems it may actually be more effective to first restrict the number of available options for a given variable, without replacing it immediately with any of them.

1.2 Towards a unified method: previous work

A particulary interesting possibility is to use both logic and mathematical modeling and attempt to draw inference from both theories. Early ideas hinting at the benefits of combining logic and mathematical programming can be traced back to the late 1960s, in the work of HAMMER AND RUDEANU[40] and later in the work of JEROSLOW[54] on logic-based decision support. Jeroslow's mixed background and interests, initially in logic and later in polyhedral theory, played a key role in his pathbreaking work towards the application of mathematical programming techniques in artificial intelligence and viceversa. He was the first to articulate the vision that we could create more effective systems, by exploiting the potential of polyhedral techniques to assist in logic based approaches and, conversely, the potential for results in applied logic to be relevant in mixed integer programming.

The research community followed suit and in the 1990s we witnessed an abundance of research on the path of integration of logic methods with optimization, much of which was captured in two books by CHANDRU AND HOOKER[28], HOOKER[47], a paper collection by MILANO[65] and most recently another, more crystallized book by HOOKER[50]. Not only did the research community realize that Jeroslow's vision was of great practical interest, but it became apparent that logic and optimization methods essentially share the same ingredients. The names of these ingredients vary from author to author, such as *branch* and *infer* in BOCKMAYR AND KASPER[23], *branch* and *bound* in LAND AND DOIG[60], *branch* and *cut* in CAPRARA AND FISCHETTI[25], *branch* and *relax*, *branch* and *price* in BARNHART ET AL[17], *branch* and *prune* in VAN HENTENRYCK[73] and VU ET AL[74], *augment* and *branch* and *cut* in LETCHFORD AND LODI[61], *search* and *infer* and *relax* in HOOKER[48], ARON, HOOKER AND YUNES[6], HOOKER[50], etc. At their core, however, these methods employ a common solution process, which relies on two basic components: (1) one *driving* the search (branching in most of them), and (2) one[2] *restricting* the search (cutting planes, relaxations, bounding techniques, pruning techniques, variable pricing and so on).

[2]Some of them break up this component into various tasks, such as *cut-and-price*, or *infer-and-relax*.

1.2.1 General purpose solvers

The first general method based on a combination of search and inference is due to Land and Doig[60], who proposed what became known as the *branch-and-bound* method for solving integer programs, as an alternative to the apparently more sophisticated cutting plane method of Gomory[38]. This was a method restricted to mixed integer programming at the time, but when looked at carefully, it can be easily seen as being a particular case of a more general method. In fact, many of the existing techniques enumerated above, and not just branch and bound, are particular cases of a more general scheme, as pointed out by Hooker[47]. This recognition has led to several attempts to bring these ideas to life and prove that unified systems can be as useful and powerful as various specialized methods when implemented in a carefully engineered computer software. A majority of these attempts focused on particular applications and were therefore typically restricted to them. They include applications such as matching in Focacci et al[35], traveling salesman in Focacci et al[33] or scheduling in Refalo[19] and Maravelias and Grossmann[62]. A few, however, were geared towards producing general purpose solvers, by focusing on the common aspects of the methods. These include the branch and cut framework ABACUS[56], the hybrid CP-IP solver ECLiPSe[2], the IP-CP solver SCIP[1] and the CP-IP-NLP metasolver SIMPL[6].

1.2.2 Modeling languages

In order for the integration to be effective, general purpose solvers need to rely on an expressive enough language, which can provide the modeler with sufficient opportunities to reveal problem structure to the underlying inference engine. The most successful existing implementations of integrated methods are accompanied by fairly rich languages, some of which even provide mechanisms for specifying how the search should be conducted, not only how inference should be drawn.[3]

[3]Such languages are called procedural, because the modeler can provide, to some extent, a procedure, or recipe on how the problem is to be solved. They are clearly more powerful, but require more expertise from the modeler, than the declarative languages, which only allow the specification of the problem, not of the search procedure.

Among the most well known modeling languages for optimization we count, for instance, OPL[44], which provides the modeling layer for the ILOG optimization suite, namely the mixed integer programming solver ILOG CPLEX[52] and the constraint programming solver ILOG Solver[51]. A similar language, Mosel[29], comes with the Dash Optimization suite, again a combination of several separate solvers: one for mixed integer programming, Xpress-Optimizer[31], one for constraint programming, Xpress-Kalis[32], as well as two more recent solvers for stochastic and nonlinear programming. Several other languages have been created in the 1990s for various special purpose solvers. Examples include Helios[45], Newton[46] and Numerica[43], which were tailored for global optimization with nonlinear constraints using interval analysis, AMPL[36], GAMS[24] and AIMMS[22] for mathematical programming and PLAM[18] for algebraic modeling of integer programs and constraint logic programs.

Several of these languages support ideas related to the integration of optimization methods, some to a larger extent than others. Even though it would be possible to create hybrid models in these languages, this task is not always simple, since in fact none of them was created with the stated goal of bringing together multiple methods under one roof, at least not at the level where inference and search interact. The modeling aspect of optimization is an ongoing research issue and it is reasonable to expect that languages of the future would pay significantly more attention to the integration process, especially that we can already see such integrated methods being used with increasing frequency.

1.3 Organization

The material in this thesis is organized as follows. Chapter 2 introduces the notion of reformulation and establishes its connection with logical inference. Chapter 3 then presents an overview of integer programming and constraint programming, with an in-depth discussion of the pervasive disjunctive constraints and techniques for reformulating them as linear inequalities and using the derived inequalities for inference. Chapter 4 describes a unifying modeling and solution framework that can immediately incorporate linear programming, integer programming, disjunctive programming and constraint programming.

The key concept of this framework is that of metaconstraints, which can be equipped with reformulation rules so that inference is derived using multiple theories at each step of the process. The modeling concepts and solution process of this framework are illustrated using three application examples: production planning, product configuration and machine scheduling. Finally, Chapter 5 presents a more in-depth discussion of a robust optimization problem, which emphasizes the advantages of modeling using higher level metaconstraints. The results shown at the end of this chapter compare the performance of a constraint programming approach to that of a pure mixed integer approach. They show that the additional inference obtained by using graph theoretical arguments at each step of the solution process is instrumental in obtaining results for larger problem instances, as well as getting significantly faster results on small instances.

Chapter 2

Inference via reformulation

It is often the case that in order to prove that a given statement is false, it may be useful to restate it in a different language and use different types of arguments. Such a *reformulation*, which is in essence a syntactic translation, cannot be done in arbitrary ways. It must be done such that the resulting statement is always an *implication*, if not an equivalence, of the original. The relaxation techniques used in optimization are perfect examples of syntatic translations which result in such an implication. If the implication is then found to be false, we can infer by application of the *modus tollens* rule that the original statement is false too.

2.1 The importance of reformulation

Reformulation is a valuable tool in problem solving, whose role cannot be understated. To illustrate its power, let us look at an instructive example, the pigeonhole problem, adapted from McAloon et al[64]. Further examples of inference via reformulation will be discussed later in the context of disjunctive programming, where we look at two important reformulation techniques: conjunction elimination from systems of linear inequalities (known as the derivation of surrogates) and disjunction elimination from systems of linear inequalities (known as the derivation of disjunctive cuts). In the pigeonhole problem, we have m pigeons and must place them in n holes such that only one pigeon is in any one hole. The problem can be easily modeled using the language of *propositional logic*, and

we could attempt to solve it within that theory, but as it will become clear, the solution process (proof) can be substantially shortened if we use reformulation.

A propositional logic model

Let p_{ij} be a variable with possible values $\{true, false\}$ that indicates whether the ith pigeon is in the jth hole. Then

$$\bigwedge_i \bigvee_j p_{ij} \tag{2.1}$$

states that "every pigeon has to be placed in (at least) one hole", and

$$\bigwedge_j \bigwedge_i \left(p_{ij} \rightarrow \bigwedge_{k<i} \neg p_{kj} \right) \tag{2.2}$$

states that "any hole can contain at most one pigeon". Note that we are not able to state the condition "*at most one* pigeon can be in hole j", using a single formula for each j, since propositional logic does not have a connective for this.

If we now step back and read the description of the problem again, we can immediately see that the problem admits a solution if and only if $m \leq n$. We know this because we subconsciously use some additional arguments, namely *counting arguments*, which are actually not part of the propositional logic theory. In particular, we use knowledge about the "sum" $(+)$ and "greater than" $(>)$ operators: we add up the pigeons, then the holes, then we compare the resulting values and decide that the problem is unsolvable if there are more pigeons than holes. However, without these two operators, classical theorem provers may find it very hard to prove that no solution exists in the case when $m > n$. The shortest resolution proof that detects the unsatifiability of this problem involves a number of steps that is exponential in n, as shown in HAKEN[39].

Since it appears that counting arguments can be quite effective in *detecting the infeasibility* of this problem, let us attempt to *translate* this propositional model into a theory which allows such arguments. Integer 0-1 programming is a good candidate, because we have a set of simple rules which can translate from proposition logic syntax to 0-1 integer programming syntax (see, for example, HAMMER AND RUDEANU[40]).

A 0-1 integer programming reformulation

Let x_{ij} be a variable with possible values $\{0, 1\}$, such that $x_{ij} = 1$ iff $p_{ij} = true$ and $x_{ij} = 0$ iff $p_{ij} = false$. Using these variable translation rules, it is easy to verify that the propositinal logic condition

$$\bigvee_j p_{ij} \tag{2.3}$$

implies the 0-1 integer programming condition

$$\sum_j x_{ij} \geq 1 \tag{2.4}$$

Thus, we can deduce the following:

$$\bigwedge_i \bigvee_j p_{ij} \rightarrow \bigwedge_i \sum_j x_{ij} \geq 1 \rightarrow \left(\sum_j x_{ij} \geq 1, \forall i \right) \tag{2.5}$$

which essentially translates the statement that "every pigeon has to be placed in (at least) one hole" from propositional logic to 0-1 integer programming.

In order to translate the second condition, let us first rewrite it as a conjunction of disjunctions, so that we can apply the same translation rule as above. This is known as writing the formula in *conjunctive normal form* (CNF). To obtain this form, we use the following equivalences, which are easily obtained starting from the axioms of propositional calculus and applying the modus ponens inference rule:

$$(A \rightarrow B) \leftrightarrow (\neg A \vee B) \tag{2.6}$$

and

$$A \vee (B \wedge C) \leftrightarrow (A \wedge B) \vee (A \wedge C) \tag{2.7}$$

Now we can write

$$\bigwedge_j \bigwedge_i \left(p_{ij} \rightarrow \bigwedge_{k<i} \neg p_{kj} \right) \rightarrow \bigwedge_j \bigwedge_i \left(\neg p_{ij} \bigvee \bigwedge_{k<i} \neg p_{kj} \right) \rightarrow \bigwedge_j \bigwedge_i \bigwedge_{k<i} (\neg p_{ij} \vee \neg p_{kj})$$

The variable mapping we established above allows us to say that $\neg p_{ij}$ is true iff $1 - x_{ij} = 1$, so we can now rewrite the condition that any hole can contain at most one pigeon using

$O(nm^2)$ linear inequalities:

$$(1 - x_{ij}) + (1 - x_{kj}) \geq 1, \forall j, \forall i, \forall k < i \tag{2.8}$$

A linear programming reformulation

In this form, the problem still remains quite difficult to solve for integer programming systems, even for small values of $m > n$. However, we are now able to express the condition that at most one pigeon can be in hole j quite naturally with a single formula for each j:

$$\sum_i x_{ij} \leq 1, \forall j \tag{2.9}$$

which we can use in place of the above $O(nm^2)$ inequalities. Together with the condition that each pigeon be placed in at least one hole,

$$\sum_j x_{ij} \geq 1, \forall i \tag{2.10}$$

and with a *further reformulation* of the x variables into continuous y variables, we obtain the linear program

$$\sum_i y_{ij} \leq 1, \forall j$$
$$\sum_j y_{ij} \geq 1, \forall i$$

whose feasible region is the empty set if $m > n$.

Thus, the fact that the problem does not have any solutions for $m > n$ can be established *without performing any search* by working with an appropriate theory: in this case, the theory of continuous linear programming.

2.2 A special case: relaxation

In general, reformulation does not have to be done for the entire problem, and even when it is done, individual constraints could be reformulated in different theories. However, it is often the case that in practice all (or a subset of) the constraints are reformulated in the same target language, leading to what is usually referred to as a *relaxation* of the

original problem. This, for example, is the way integer programming solvers approach the inference step of the solution process. The use of discrete variables introduces discontinuities and nonconvexity, which makes it considerably more difficult to reason about them than it would be to reason about continuous variables. Therefore the original constraints (including the integrality constraints) are reformulated into linear inequalities (the integrality constraints $x \in D_x \subseteq \mathbf{Z}$ are reformulated into the conjunction of linear inequalities $\inf\{D_x\} \leq x \bigwedge x \leq \sup\{D_x\}$) and then the simpler linear programming theory can be used to derive inference. Such inference comes in various forms, but serves the two basic roles we mentioned earlier: (a) refutation of the statement represented by the current search state, or (b) guidance for restricting and continuing the search.

Using relaxation to detect infeasibility

A continuous view of the problem can provide an easy proof that the search has gone wrong, if it detects that its own feasible region is empty. Secondly, if the region is not empty, nothing can be said about the feasibility of original problem, but due to the special properties of the continuous relaxation, we might be able to say something about its optimality (which could also be regarded as feasibility). Thus, if the original problem includes an objective function in its statement, the continuous relaxation could still help us decide that there is no point in continuing with this search direction, if we can show that there could be no solution inside the continuous relaxation which provides a better value than whatever best solution we may have found previously. This conclusion of infeasibility can be reached in two ways: either the best solution of the continuous relaxation is *already* worse than our best discrete solution, or we could show that it *would become* worse no matter what search decisions we make from here. The first is easy to check, by simply evaluating the objective function on the most extreme point of the continuous feasible region. The second involves evaluating the changes in the objective function by plugging in some of the values for the original variables which the search has not yet explored. Values which would make the objective worse than the best discrete solution so far can be safely discarded. A similar technique is used in constraint programming, where it is known as *reduced-cost based filtering* (see FOCACCI ET AL[34]).

Using relaxation to avoid infeasibility

Note here the second role played by the continuous view of the problem, that of prevention, or guidance: even if we are unable to refute the current state, because perhaps we cannot rule out all remaining values using the objective function, we can still safely remove *some of them*, essentially reducing the number of (bad) choices the search would have to try from this point.

An additional (and particularly useful) preventive inference technique based on a continuous view of the problem involves *cutting planes*. These are linear inequalities meant to *help the bounding process* through which we attempt to detect infeasible (sub-optimal) search states, as indicated before. Cutting planes can be viewed as being of two types, depending on how they are inferred: (1) either directly from the problem, or (2) from an implication of the problem (which could be the continuous linear relaxation). In the first case, they are called *structural cuts* (or sometimes polyhedral cuts), because they are implied by the structure of the problem and depend neither on the language in which the problem is stated, nor on the search state. These cuts are *direct linear implications* of the original problem. The well-known comb inequalities for the traveling salesman problem (see Applegate et al[3]) are a perfect example of such inequalities. The second type of cuts are inferred from the continuous view of the problem, by focusing on removing a tentative solution if it fails to satisfy all the constraints of the original problem. They are derived *at some stage during the search*, using two basic inference rules: (1) *non-negative combinations of linear inequalities* (this is where the continuous view plays a role) and (2) *integer rounding* (this is where the original integrality constraints play a role). As such, these cuts are *indirect* linear implications of the problem and are valid only during subsequent stages of the search (although they can be made valid everywhere through a process called lifting). Examples of such cuts include Gomory cuts (see Gomory[38]), mixed integer rounding cuts (see Marchand et al[63]), intersection cuts (see Balas[9]) and disjunctive cuts (see Balas[10]). We will have a chance to look at this type of cutting planes in more detail in Section 3.1.3 when we discuss inference based on a special subclass of disjunctive cuts, namely lift-and-project cuts (see Balas[7]).

Chapter 3

Optimization theories

In this chapter we review two commonly used optimzation theories: *integer programming* (IP) and *constraint programming* (CP). They constitute the starting point for the unifying framework presented in Chapter 4, since they share many features in terms of solution method, but at the same time complement each other in terms of inference capabilities. Even though the scope of the framework we propose is in no way limited to these two theories, we restrict the discussion to them for the following reasons: (1) they are both well studied and understood, (2) they have been successfully applied to a wide range of practical applications, (3) they are among the most frequently used theories in various hybrid solutions proposed in the last decade and (3) our implementation of this framework is currently limited[1] to these two theories.

3.1 Integer and disjunctive programming

Integer programming is linear programming with the additional constraints that some of the variables must take integer (discrete) values. For this reason, integer programs are typically much more difficult to solve than pure linear programs. It is interesting to point out here that it was unknown for a long time whether even solving a pure linear program can be done efficiently. The original proof proposed by DANZIG[30] in 1947, based on linear

[1] This limitation refers to available and tested functionality.

algebraic arguments, required an exponential number of steps in the worst case, as shown by Klee and Minty[58]. Three decades later, however, the soviet mathematician Leonid Khachyan[57] was able to show that linear programming does admit a polynomial length proof, but for that he had to use arguments from mathematical analysis, as opposed to just linear algebra. No such proof has been found for integer programming, and it is widely believed that none exists.

Disjunctive programming goes one step further from integer programming, by allowing in the language an additional type of constraints, namely disjunctive (linear) constraints. These constraints naturally capture the idea of choice, which is inherent to all combinatorial optimization problems. Integer programs, pure and mixed, and a host of other non-convex programming problems can be stated more naturally as linear programs with logical conditions, more precisely statements about linear inequalities involving the operators "or". For a good introduction to disjunctive programming see Balas[11].

Modeling and solution process

Integer programming solvers typically solve a linear program at every node of the search tree in order to obtain bounds on the objective function (and sometimes with the hope that the resulting solution is also feasible with respect to the integrality constraints). There are no integer programming solvers that deal directly with disjunctive constraints at the modeling level. Instead, these constraints are either enforced implicitly, through a branching mechanism, or reformulated (relaxed) into linear constraints that integer solvers can deal with. In the remaining of this section, we focus on this reformulation and present one type of linear inequalities (cuts) that can be derived from disjunctions and can be used as an inference tool in an integrated solver. The algorithm discussed here is an extension of the one created by Balas and Perregaard[8], and it is the latest development in a field of research that originated in the work on intersection cuts (also known as convexity cuts) by Young[77], Balas[9], Glover[37] and Owen[67].

3.1.1 Disjunctive linear constraints

Consider the family of inequalities $\{ a^i x \geq b^i \}_{i \in I}$, with $x \in \mathbf{R}_+, a^i \in \mathbf{R}, b^i \in \mathbf{R}$. A *linear disjunction* (or simply, disjunction) is a proposition which states that "*at least one of the inequalities in this family is satisfied*" and has the form:

$$\bigvee_{i \in I} a^i x \geq b^i \tag{3.1}$$

Note that when $b^i > 0$, which is always the case when we derive cuts from a (feasible basic solution in a) simplex tableau, disjunction (3.1) is clearly equivalent with one in which we multiply each disjunct with the inverse of its right hand side:

$$\bigvee_{i \in I} a^i x \geq b^i \Leftrightarrow \bigvee_{i \in I} \frac{a^i}{b^i} x \geq 1 \tag{3.2}$$

It is often convenient to represent a disjunction in the form (3.2) whenever $b^i > 0$, because it allows us to compare the coefficients (and hence the strength) of inferred inequalities one by one without having to worry about the right hand side. We say that disjunctions in this form, as well as the inequalities implied by them, are *normalized*. We will later see that it is also desirable, not just convenient, to represent disjunctions in normalized form.

More generally, we can define disjunctions for *systems* (conjunctions) of linear inequalities

$$\bigvee_{i \in I} \mathbf{A}^i \mathbf{x} \geq \mathbf{b}^i \tag{3.3}$$

where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{b}^i \in \mathbf{R}^m$, $\mathbf{x} \in \mathbf{R}^n$, with the important special case $m = 1$

$$\bigvee_{i \in I} \mathbf{a}^i \mathbf{x} \geq b^i \tag{3.4}$$

with $\mathbf{a}^i \in \mathbf{R}^n$. Note also that the normalization argument used in (3.2) can be easily generalized to apply to these cases as well.

Disjunctive programming is closely related to linear programming, since a sentence involving the disjunctive connective "$\vee$" can be replaced by an implication using only linear connectives. This is a *reformulation* process, and it can help detecting when a disjunctive model is infeasible.

3.1.2 Inference via reformulation

We now discuss the basic translation rules that allow us to convert a disjunction into a linear inequality. By conversion, it is understood that we want to derive an inequality such that all solutions that satisfy the original disjunction, also satisfy that inequality. In other words, the derived inequality should be an *implication* of the disjunction, and as such, if it turns out to be a false statement, then we can say the same about the disjunction.

Linear implications that eliminate disjunction

The following basic translation rules provide the main instrument used to derive a class of linear inequalities known as *disjunctive cuts* (see BALAS[11, 12]). These inequalities are reformulations of the original disjunction, into a language from which the "$\vee$" connective has been eliminated, and as such, they can be reasoned about using the simpler theory of linear programming.

Proposition 3.1.1. *If the disjunction* $\bigvee_{i \in I} a^i x \geq b^i$ *is true, then the following linear inequality must hold:*

$$\max_i \{a^i\} x \geq \min_i \{b^i\} \tag{3.5}$$

Similarly, if $\bigvee_{i \in I} a^i x \leq b^i$ *is true, then*

$$\min_i \{a^i\} x \leq \max_i \{b^i\} \tag{3.6}$$

is also true.

These rules can be easily generalized to the case when a and x are n-dimensional, as well as to disjunctions of systems of linear inequalities, such as (3.3) and (3.4). .

Proposition 3.1.2. *Let* $\mathbf{A}^i$ *be an* $m \times n$ *matrix,* $\mathbf{a}^i \in \mathbf{R}^n$, $\mathbf{b}^i \in \mathbf{R}^m$, *for any* $i \in I$, *and* $\mathbf{x} \in \mathbf{R}^n_+$. *Then the following implications hold:*

$$\bigvee_{i \in I} \mathbf{a}^i \mathbf{x} \geq b^i \rightarrow \sum_{j=1}^{n} \max_i \{a^i_j\} x_j \geq \min_i \{b^i\} \tag{3.7}$$

$$\bigvee_{i \in I} \mathbf{a}^i \mathbf{x} \leq b^i \rightarrow \sum_{j=1}^{n} \min_i \{a^i_j\} x_j \leq \max_i \{b^i\} \tag{3.8}$$

$$\bigvee_{i \in I} \mathbf{A}^i\mathbf{x} \geq \mathbf{b}^i \rightarrow \sum_{j=1}^{n} \max_i\{\overline{\mathbf{A}}_j^i\}x_j \geq \min_i\{\underline{\mathbf{b}}^i\} \tag{3.9}$$

$$\bigvee_{i \in I} \mathbf{A}^i\mathbf{x} \leq \mathbf{b}^i \rightarrow \sum_{j=1}^{n} \min_i\{\underline{\mathbf{A}}_j^i\}x_j \leq \max_i\{\overline{\mathbf{b}}^i\} \tag{3.10}$$

where $\mathbf{A}_j^i$ represents the jth column of matrix $\mathbf{A}^i$ and $\overline{\mathbf{v}}$ (respectivelly $\underline{\mathbf{v}}$) represents the largest (respectivelly smallest) element of vector $\mathbf{v}$.

The reformulation rules in Proposition 3.1.2 are of central importance in disjunctive (and therefore integer) programming, because they can be used to derive a large family of cutting planes, including ones for mixed integer programming, such as mixed integer Gomory cuts and mixed integer rounding (MIR) cuts (see NEMHAUSER AND WOLSEY[66] and MARCHAND AND WOLSEY[63]), where the Chvatal-Gomory rounding procedure (see SCHRIJVER[70]) fails. The consequents of these implications are called *disjunctive cuts* (see BALAS[11]).

Linear implications that eliminate conjunction

The other reformulation rule which plays a key role in integer and linear programming is the following:

Proposition 3.1.3. *Let $\mathbf{A}$ be an $m \times n$ matrix, $\mathbf{b} \in \mathbf{R}^m$ and $\mathbf{x} \in \mathbf{R}_+^n$. Then the following implication holds:*

$$(\mathbf{Ax} \geq \mathbf{b} \wedge \mathbf{u} \geq \mathbf{0}) \rightarrow \mathbf{uAx} \geq \mathbf{ub} \tag{3.11}$$

This property allows us to summarize an *entire system* (conjunction) of inequalities with a *single* inequality, which is much easier to check against infeasibility. The above implication is clearly a reformulation, since it removes from the language the conjunction connective "$\wedge$", and therefore allows us to use a simpler theory in order to detect if the original system is inconsistent. The resulting inequality, $\mathbf{uAx} \geq \mathbf{ub}$, is called a *surrogate*. It is used to derive so-called *valid inequalities* for the linear programming relaxation of an integer program.

Definition 3.1.4 (Valid inequality)**.** An inequality $\alpha\mathbf{x} \geq \beta$ is said to be *valid* for a set $S \subseteq \mathbf{R}^n$ if it is satisfied by all the points $\mathbf{x} \in S$.

Clearly, all surrogates $\mathbf{uAx} \geq \mathbf{ub}$ are valid for the set $P = \left\{ \mathbf{x} \in \mathbf{R}^n_+ : \mathbf{Ax} \geq \mathbf{b} \right\}$. The requirement that $\mathbf{u} \geq \mathbf{0}$ ensures that the sense of the inequalities is preserved. Obviously, this requirement can be relaxed to $u_i \in \mathbf{R}$ if the ith inequality of the system $\mathbf{Ax} \geq \mathbf{b}$ is in fact an equality (because a negative multiplier would not affect its sense).

Stronger inference via a combination of these rules

If we now look back at the linear inequalities obtained through Propositions 3.1.1 and 3.1.2, it is not difficult to see that they are not the strongest inferences we can make from a disjunction of linear inequalities. Indeed, let us take a very simple example, to illustrate what can be done in order to obtain a stronger inequality from such a disjunction. Consider the disjunction:

$$4x \geq 3 \vee x \geq 1 \tag{3.12}$$

If we simply apply Proposition 3.1.1 to this disjunction, we obtain the valid inequality $4x \geq 1$. However, we can do better. Notice that if we multiply the second disjunct with 3, we obtain an *equivalent* disjunction:

$$4x \geq 3 \vee 3x \geq 3 \tag{3.13}$$

But the inequality we can now infer from this disjunction by applying Proposition 3.1.1 is much stronger: $4x \geq 3$. This is so because it has the same left hand side, but a tighter right hand side (that is, there are fewer values of x that satisfy it). We say that this inequality *dominates* the one we had obtained before.

Definition 3.1.5 (Dominating inequality)**.** An inequality $\alpha\mathbf{x} \geq \beta$ is said to *dominate* another inequality $\alpha'\mathbf{x} \geq \beta'$ if $\alpha \leq \alpha'$ and $\beta \geq \beta'$.

What did actually happen in the transformation from (3.12) to (3.13)? We managed to infer a tighter valid inequality by:

1. collecting together all the inequalities of the disjuncts into a system (*conjunction*) of inequalities:

$$\begin{pmatrix} 4x & \geq & 3 \\ x & \geq & 1 \end{pmatrix}$$

2. then carefully choosing a non-negative multiplier $\mathbf{u} = (u_1, u_2) = (1, 3)$ for this system such that to preserve (or decrease) the maximum coefficient of x in the left hand side, and to increase (or preserve) the minimum value on the right hand side:

$$\begin{pmatrix} 4x & \geq & 3 & | & \times 1 \\ x & \geq & 1 & | & \times 3 \end{pmatrix} \Leftrightarrow \begin{pmatrix} 4x & \geq & 3 \\ 3x & \geq & 3 \end{pmatrix}$$

3. and finally applying disjunction elimination to the resulting inequalities:

$$(4x \geq 3 \vee 3x \geq 3) \rightarrow 4x \geq 3 \tag{3.14}$$

Note that when we derived the weaker $4x \geq 1$ from the original disjunction, we in fact used a default multiplier, namely $\mathbf{u} = (u_1, u_2) = (1, 1)$. The new multiplier is certainly better, and we can tell that because it reduced the difference between the left hand side and the right hand side of the resulting inequality. The fact that we were able to increase the right hand side means that the original inequality, although valid, was not *maximal*.

Definition 3.1.6 (Maximal inequality)**.** A *maximal* valid inequality is one which is not dominated by any other valid inequality.

Such inequalities are also known as *face defining*, or *supporting* inequalities. Our goal is to devise a technique to find valid inequalities that are as strong as possible (maximal).

Remark 3.1.7. We said before that it is often convenient to normalize the disjunction when the right hand side is strictly positive. It should be clear by now that it is not just convenient. Note that the effect of this normalization is the same as what we have done with the multiplier $\mathbf{u}$ in order to strengthen the inferred inequality. Indeed, had we first normalized the disjunction, we would have obtained:

$$\frac{4}{3}x \geq 1 \vee x \geq 1$$

which now leads directly (by Proposition 3.1.2) to the stronger valid inequality (3.14). So it is always a good idea to normalize disjunctions that satisfy this condition.

Connection with linear programming duality

You might have recognized already the principle at work here (and perhaps the connection with linear programming duality). Indeed, in linear programming, we try to find a non-negative multiplier $\mathbf{u}$ for a system of inequalities $\mathbf{Ax} \geq \mathbf{b}$ which transforms it into an implied inequality $\mathbf{uAx} \geq \mathbf{ub}$, such that we keep $\mathbf{uA}$ below some predetermined objective vector $\mathbf{c}$ (i.e. $\mathbf{uA} \leq \mathbf{c}$) and at the same time increase the right hand side $\mathbf{ub}$ by as much as possible. We can find such a multiplier $\mathbf{u}$ by solving a linear programming dual of the original LP:

LP	Dual
$\min \quad \mathbf{cx}$	$\max \quad \mathbf{ub}$
$s.t \quad \mathbf{Ax} \geq \mathbf{b}$	$s.t \quad \mathbf{uA} \leq \mathbf{c}$
$\mathbf{x} \geq \mathbf{0}$	$\mathbf{u} \geq \mathbf{0}$

There is a difference, however, with finding the multiplier $\mathbf{u}$ that strengthens an inequality inferred from a disjunction of single linear inequalities $\mathbf{a}^i\mathbf{x} \geq b^i$ (collected into a system of inequalities $\mathbf{Ax} \geq \mathbf{b}$). The difference is that we are not computing the dot products $\mathbf{uA}_j$ and $\mathbf{ub}$, as we do in the dual linear program, but instead we take the maximum and minimum of the element-wise products $\mathbf{u} \odot \mathbf{A}$ and $\mathbf{u} \odot \mathbf{b}$, where:

$$\mathbf{u} \odot \mathbf{A} = \begin{pmatrix} u_1 a_{11} & u_1 a_{12} & \dots & u_1 a_{1n} \\ u_2 a_{21} & u_2 a_{22} & \dots & u_2 a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ u_m a_{m1} & u_m a_{m2} & \dots & u_m a_{mn} \end{pmatrix}, \mathbf{u} \odot \mathbf{b} = \begin{pmatrix} u_1 b_1 \\ u_2 b_2 \\ \vdots \\ u_m b_m \end{pmatrix}$$

So if we denote

$$\begin{aligned} \alpha &= \overline{\mathbf{u} \odot \mathbf{A}} &= \left(\max_i \{ u_i a_{i1} \}, \max_i \{ u_i a_{i2} \}, \dots, \max_i \{ u_i a_{in} \}\right) \in \mathbf{R}^n \\ \beta &= \underline{\mathbf{u} \odot \mathbf{b}} &= \min_i \{ u_i b_i \} \in \mathbf{R} \end{aligned}$$

then finding the strongest inferred inequality $\alpha\mathbf{x} \geq \beta$ amounts to solving the following problem:

$$\begin{aligned} \max \quad & \beta \\ s.t \quad & \alpha \leq \mathbf{c} \\ & \mathbf{u} \geq \mathbf{0} \end{aligned} \tag{3.15}$$

where the unknown is the multiplier $\mathbf{u}$ and where the bounding vector $\mathbf{c} = \overline{\mathbf{1} \odot \mathbf{A}}$ contains the coefficients of the inequality obtained using the default multiplier $\mathbf{u} = \mathbf{1}$.

Finding valid inequalities for a disjunction of conjunctions

It is easy to verify that an inequality $\alpha\mathbf{x} \geq \beta$ is valid for the set $P = \left\{ \mathbf{x} \in \mathbf{R}^n_+ : \mathbf{A}\mathbf{x} \geq \mathbf{b} \right\}$ if and only if it is dominated by a surrogate of $\mathbf{A}\mathbf{x} \geq \mathbf{b}$. Furthermore, if we have a disjunction of such sets, written as

$$\bigvee_{i \in I} \mathbf{x} \in P^i \Leftrightarrow \bigvee_{i \in I} \mathbf{A}^i\mathbf{x} \geq \mathbf{b}^i \tag{3.16}$$

a theorem by Balas[11] says that $\alpha\mathbf{x} \geq \beta$ is valid for (3.16) if an only if it is dominated by a surrogate $\mathbf{u}^i\mathbf{A}^i\mathbf{x} \geq \mathbf{u}^i\mathbf{b}^i$ of each of the disjuncts $i \in I$. In other words, if and only if it is valid for each of the sets $P^i = \left\{ \mathbf{x} \in \mathbf{R}^n_+ : \mathbf{A}^i\mathbf{x} \geq \mathbf{b}^i \right\}$. So, in order to find a valid inequality for (3.16), it is enough to find a set of multipliers $\mathbf{u}^i$ with this property, and then use Proposition 3.1.2. We obtain the following valid inequality:

$$\sum_j \max_{i \in I} \left\{ \mathbf{u}^i\mathbf{A}^i_j \right\} x_j \geq \min_{i \in I} \left\{ \mathbf{u}^i\mathbf{b}^i \right\} \tag{3.17}$$

To find these multipliers, we can set up a simple linear program and require that they define inequalities $\mathbf{u}^i\mathbf{A}^i\mathbf{x} \geq \mathbf{u}^i\mathbf{b}^i$ which dominate some $\alpha\mathbf{x} \geq \beta$:

$$\begin{array}{lrcll} \max & 0 & & & \\ s.t & \alpha & \geq & \mathbf{u}^i\mathbf{A}^i & \forall i \in I \\ & \beta & \leq & \mathbf{u}^i\mathbf{b}^i & \forall i \in I \\ & \mathbf{u}^i & \geq & \mathbf{0} & \forall i \in I \\ & \alpha, \beta & \in & \mathbf{R}^n & \end{array} \tag{3.18}$$

The unknowns in this system are $\mathbf{u}^i$, α and β. The first two inequalities simply state that the surrogates $\mathbf{u}^i\mathbf{A}^i\mathbf{x} \geq \mathbf{u}^i\mathbf{b}^i$, for all $i \in I$, dominate $\alpha\mathbf{x} \geq \beta$, which implies that $\alpha\mathbf{x} \geq \beta$ will be valid for (3.16). Solving this system for $\mathbf{u}$ (α and β being unconstrained in sign, can be eliminated), we obtain the coefficients of valid inequalities of the form $\alpha\mathbf{x} \geq \beta$ that obviously satisfy:

$$\alpha_j = \max_{i \in I} \left\{ \mathbf{u}^i\mathbf{A}^i_j \right\}$$

and

$$\beta = \min_{i \in I} \{ \mathbf{u}^i \mathbf{b}^i \}$$

In other words, we have obtained the same inequality as we would (if we knew $\mathbf{u}^i$) using the reformulation rules in Proposition 3.1.2, namely (3.17).

Finding stronger inequalities

We obviously wrote (3.18) such that it finds *one* set of multipliers $\mathbf{u}^i$ in order to obtain a valid inequality. We saw before that it may be possible to find stronger inequalities, by finding *better* multipilers. In order to do this, note first that we can rewrite the system (3.18), by moving all the variables on the left hand side, in the following form:

$$\begin{array}{rrcll} \max & 0 & & & \\ s.t & \alpha - \mathbf{u}^i \mathbf{A}^i & \geq & 0 & \forall i \in I \\ & -\beta + \mathbf{u}^i \mathbf{b}^i & \geq & 0 & \forall i \in I \\ & \mathbf{u}^i & \geq & \mathbf{0} & \forall i \in I \\ & \alpha, \beta & \in & \mathbf{R}^n & \end{array} \tag{3.19}$$

which makes it apparent that this is a polyhedral cone. This observation becomes useful if we want to optimize a more interesting function than $\max 0$, one which could involve α and β.[2] It tells us that we would have to truncate this cone with some bounding inequality (or inequalities) in order to prevent α and β from scaling up indefinitely. Evidently, the choice of the bounding constraints used to truncate the cone could have an impact on the type (and quality) of solutions we obtain, so it is important to pay attention to this aspect.

Balas et al[14] consider adding one of three possible bounding constraints (which they call normalizations) to the system (3.19) in order to guarantee that optimal solutions exist when the objective function involves the unknowns α and β (i.e. the coefficients of the valid inequality). Their bounding constraints focused on one of these variables only, and they were: (i) $|\beta| \leq 1$, or (ii) $|\alpha_j| \leq 1, \forall j$, or (iii) $\sum_j |\alpha_j| \leq 1$. Clearly, by imposing such constraints, we limit the class of inequalities that we can obtain with (3.19), but at least

[2] Therefore it could help us find better valid inequalities $\alpha\mathbf{x} \geq \beta$, in some sense.

we hope that the system has a (finite, nontrivial) solution. In the computational study of the original lift-and-project method by BALAS ET AL[15], which generated cuts from 0-1 disjunctions on a single variable (i.e. $x_k \leq 0 \vee x_k \geq 1$ for some integer constrained x_k), the bounding constraint $|\beta| \leq 1$ was the best (in terms of cut quality) among the three. Unfortunately, the problem with this constraint is that α can still be unbounded. Indeed, to find a cut which is (maximally) violated at some point $\tilde{x}$, we would use an objective like $\min \alpha\tilde{x} - \beta$. But if there exists a valid cut of the form $\alpha' x \geq 0$ with $\alpha'\tilde{x} < 0$, then the objective is clearly unbounded. The other two bounding constraints pose different problems. Even though they both achieve the goal of bounding α, they either produce cuts of lesser quality (as BALAS ET AL[15] found for $|\alpha_j| \leq 1$), or they are are computationally more demanding, by increasing the size of the linear program (as it is the case with $\sum_j |\alpha_j| \leq 1$, which in order to be used inside a linear solver requires that we split the unrestricted variables α into two positive variables $\alpha = \alpha^+ - \alpha^-$).

A better bounding constraint was proposed by CERIA AND PATAKI[26]:

$$\sum_{i \in I} \sum_j u^i_j \leq 1 \tag{3.20}$$

Because the multipliers $\mathbf{u}^i$ are non-negative, and because α and β are obtained as linear combinations of $\mathbf{u}^i$, this constraint is sufficient to bound all the variables in (3.19). This and the previous three bounding constraints are discussed in detail by CERIA AND SOARES[27]. They also give some nice interpretations of the dual formulation of (3.19) when the different constraints are used.

Generating the deepest cuts

The problem of finding better valid inequalities arises when we have a certain reference point $\overline{\mathbf{x}} \in \mathbf{R}^n$ which does not satisfy disjunction (3.16). In this case, we want to derive an inequality $\alpha\mathbf{x} \geq \beta$ which is valid for the disjunction, but is *maximally* violated by this point. To find this inequality, we can use the linear program (3.19), to which we add the bounding constraint (3.20) and the objective function $\min \alpha\overline{\mathbf{x}} - \beta$. We obtain the

following linear program, which BALAS ET AL[14] call *cut generating linear program*:

$$\begin{array}{rrcll} \min & \alpha\overline{\mathbf{x}} - \beta & & & \\ s.t & \alpha - \mathbf{u}^i\mathbf{A}^i & \geq & 0 & \forall i \in I \\ & -\beta + \mathbf{u}^i\mathbf{b}^i & \geq & 0 & \forall i \in I \\ & \mathbf{u}^i & \geq & \mathbf{0} & \forall i \in I \\ & \sum_i \mathbf{u}^i \cdot \mathbf{1} & \leq & 1 & \\ & \alpha, \beta & \in & \mathbf{R}^n & \end{array} \tag{CGLP}$$

Solving this system yields an inequality $\alpha\mathbf{x} \geq \beta$, with

$$\alpha_j = \max_{i \in I} \left\{ \mathbf{u}^i\mathbf{A}^i_j \right\} \quad \text{and} \quad \beta = \min_{i \in I} \left\{ \mathbf{u}^i\mathbf{b}^i \right\} \tag{3.21}$$

where $\mathbf{A}_j$ is the jth column of $\mathbf{A}$. This inequality (cut) maximizes the amount $\beta - \alpha\overline{\mathbf{x}}$ by which $\overline{\mathbf{x}}$ is cut off (separated from) the solutions of the original disjunctive program.

Making the cuts valid for the original problem

If separating inequalities such as (3.21) are generated at nodes other than the root of a search tree, they subsume the decisions that search has made so far, by assigning values to components of $\mathbf{x}$. Those components x_j of $\mathbf{x}$ that have been fixed no longer participate actively in the selection of the dual multipliers $\mathbf{u}^i$ in (CGLP), since their corresponding columns from $\mathbf{A}^i_j$ have been moved (added as a constant factor) to the right hand side $\mathbf{b}^i$. Thus, the fact that $\alpha\mathbf{x} \geq \beta$ is dominated by the surrogates $\mathbf{u}^i\mathbf{A}^i\mathbf{x} \geq \mathbf{u}^i\mathbf{b}^i$ (which makes it a valid inequality) is likely to change when we undo those assignments (during backtracking). So these inequalities are valid only in the subtree where the assignments hold, and cannot be used in other parts of the search space.

In principle, however, it is possible to make them valid for the whole search space, by calculating appropriate values for the coefficients of the fixed variables. This process is called *lifting*. In general, this is a difficult task, which may require the solution of an integer program for every missing coefficient. Fortunately, the cuts we are looking at here have an important advantage: we can use the multipliers $\mathbf{u}$ obtained (along with the cut coefficients α and β) by solving (CGLP) to calculate, by closed form expressions, the objective coefficients α_j whose corresponding variable x_j has been fixed by search. The

expressions for these coefficients are given by:

$$\alpha_j = \max_{i \in I} \left\{ \mathbf{u}^i \mathbf{A}^i_j \right\} \tag{3.22}$$

as shown by BALAS ET AL[14, 15].

3.1.3 Lift and project as an inference technique

We now have the background necessary to discuss a more effective algorithm for generating disjunctive cuts. We focus here on lift and project cuts (see BALAS ET AL[14]), which are a special class of disjunctive cuts, derived from a two term disjunction of the form $(x_k \leq 0 \vee x_k \geq 1)$ imposed to the set $P = \left\{ \mathbf{x} \in \mathbf{R}^n_+ : \mathbf{Ax} \geq \mathbf{b} \right\}$. In other words, they are valid inequalities for the set of solutions of:

$$\begin{array}{rcl} \mathbf{Ax} & \geq & \mathbf{b} \\ \mathbf{x} & \geq & \mathbf{0} \\ x_k \leq 0 & \vee & x_k \geq 1 \end{array} \tag{3.23}$$

or, equivalently,

$$\left(\begin{array}{rcl} \mathbf{Ax} & \geq & \mathbf{b} \\ \mathbf{x} & \geq & \mathbf{0} \\ -x_k & \geq & 0 \end{array} \right) \vee \left(\begin{array}{rcl} \mathbf{Ax} & \geq & \mathbf{b} \\ \mathbf{x} & \geq & \mathbf{0} \\ x_k & \geq & 1 \end{array} \right) \tag{3.24}$$

for some $k \in \{1, \ldots, n\}$ such that $0 < \tilde{x_k} < 1$, where $\overline{x}_k$ is the value of variable x_k in the current solution of the LP relaxation. We want to impose the (nonlinear and non-convex) condition that $x_k \in \left\{ 0, 1 \right\}$. But since integer solvers cannot handle it in the model, we have to obtain a linear implication of this condition, one which is hopefully strong enough that it removes (at least) the current reference point $\overline{x}$ from further consideration.

Unstrengthened lift and project cutting planes

Note that disjunction (3.24) has the same form as the general disjunction (3.16), which means that we can find valid inequalities implied by (3.24) using the linear program defined in (3.19). In other words, we find a non-negative multiplier for each of the disjuncts, require that $\alpha\mathbf{x} \geq \beta$ be dominated by the resulting surrogates, and then check to see whether

the current solution $\overline{\mathbf{x}}$ of P violates $\alpha\mathbf{x} \geq \beta$. If it does (and it should, because we also took into account the two inequalities that forbid the current value of $\overline{x}_k$), that means we have a cut already, and we can try to find an even better one (one which $\overline{\mathbf{x}}$ violates "more") by trying a new multiplier. This, in essence, is the same procedure as the one described in BALAS ET AL[15]. They call the resulting system, together with the bounding constraint (3.20) and the objective function $\min \alpha\overline{\mathbf{x}} - \beta$ a *cut generating linear program for variable* x_k (CGLP$_k$). For each binary variable that happens to have a fractional value in the current LP solution, they solve one such system and add the resulting inequality to the problem formulation.

The cut generating linear program

Let us illustrate the process of constructing the cut generating LP and finding the right multipliers for the two disjuncts. To simplify notation, we can include the non-negativity constraints $\mathbf{x} \geq 0$ together with the rest, $\mathbf{A}\mathbf{x} \geq \mathbf{b}$, to obtain a new system $\mathbf{A}'\mathbf{x} \geq \mathbf{b}'$, where $\mathbf{A}' = \binom{\mathbf{A}}{I}$ and $\mathbf{b}' = \binom{\mathbf{b}}{0}$. Now we can write the disjunction (3.24) as

$$\begin{pmatrix} \mathbf{A}'\mathbf{x} & \geq & \mathbf{b}' \\ -x_k & \geq & 0 \end{pmatrix} \vee \begin{pmatrix} \mathbf{A}'\mathbf{x} & \geq & \mathbf{b}' \\ x_k & \geq & 1 \end{pmatrix} \tag{3.25}$$

In fact, let us make life even easier and simply drop the notation $\mathbf{A}'$ and $\mathbf{b}'$. We will use the original notation, $\mathbf{A}$ and $\mathbf{b}$, but implicitly assume that $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ includes the non-negativity constraints $\mathbf{x} \geq 0$. So (3.25) becomes simply:

$$\begin{pmatrix} \mathbf{A}\mathbf{x} & \geq & \mathbf{b} \\ -x_k & \geq & 0 \end{pmatrix} \vee \begin{pmatrix} \mathbf{A}\mathbf{x} & \geq & \mathbf{b} \\ x_k & \geq & 1 \end{pmatrix} \tag{3.26}$$

To find an inequality $\alpha\mathbf{x} \geq \beta$ which is implied by this disjunction, and at the same time is maximally violated by the current solution $\overline{\mathbf{x}}$, we need to find some corresponding non-negative multipliers $(\mathbf{u}, u) \in \mathbf{R}^m \times \mathbf{R}$ for the first disjunct

$$\begin{matrix} \mathbf{u}\times \\ u\times \end{matrix} \begin{pmatrix} \mathbf{A}\mathbf{x} & \geq & \mathbf{b} \\ -x_k & \geq & 0 \end{pmatrix} \Leftrightarrow \begin{pmatrix} \mathbf{u}\mathbf{A}\mathbf{x} & \geq & \mathbf{u}\mathbf{b} \\ -ux_k & \geq & 0 \end{pmatrix} \tag{3.27}$$

and $(\mathbf{v}, v) \in \mathbf{R}^m \times \mathbf{R}$ for the second disjunct

$$\begin{array}{r} \mathbf{v}\times \\ v\times \end{array} \begin{pmatrix} \mathbf{Ax} & \geq & \mathbf{b} \\ x_k & \geq & 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} \mathbf{vAx} & \geq & \mathbf{vb} \\ vx_k & \geq & v \end{pmatrix} \tag{3.28}$$

such that the resulting surrogates dominate $\alpha\mathbf{x} \geq \beta$. In other words, we need to solve the following linear program:

$$\begin{array}{llcrcl} \min & \alpha\overline{\mathbf{x}} - \beta & & & & \\ s.t & \alpha - \mathbf{uA} & + & u\mathbf{e_k} & \geq & 0 \\ & -\beta + \mathbf{ub} & & & \geq & 0 \\ & \alpha - \mathbf{vA} & - & v\mathbf{e_k} & \geq & 0 \\ & -\beta + \mathbf{vb} & + & v & \geq & 0 \\ & & & \mathbf{u} & \geq & \mathbf{0} \\ & \mathbf{u}\cdot\mathbf{1} + \mathbf{v}\cdot\mathbf{1} & + & u + v & \leq & 1 \\ & & & \alpha, \beta & \in & \mathbf{R}^n \end{array} \tag{CGLP$_k$}$$

where $\mathbf{e_k}$ is the kth unit vector. Note that in order to guarantee that an optimal solution to (CGLP$_k$) exists, we have also added the bounding constraint $\sum_i (u_i + v_i) + u + v \leq 1$, which is the same as (3.20) proposed by Ceria and Pataki[26].

As we have already seen in (3.21), solving this system yields a cut $\alpha\mathbf{x} \geq \beta$, with

$$\alpha_j = \begin{cases} \max\{ \ \mathbf{uA}_j \ , \ \mathbf{vA}_j \ \} & j \neq k \\ \max\{ \ \mathbf{uA}_j - u \ , \ \mathbf{vA}_j + v \ \} & j = k \end{cases}$$

and

$$\beta = \min\{\mathbf{ub}, \mathbf{vb} + v\}$$

where $\mathbf{A}_j$ is the jth column of matrix $\mathbf{A}$.

Strengthening cuts using integrality conditions

We have so far focused on strenthening the derived inequality by finding better multipliers $\mathbf{u}$ using the linear program (CGLP). However, a further strengthening is possible, by exploiting the integrality conditions on variables. For instance, inequality (3.21) can be

further strengthened using this technique, as shown by BALAS AND JEROSLOW[16]. This can be done by replacing each coefficient α_j with

$$\alpha'_j = \min\{\,\mathbf{u}\mathbf{A}_j + u_0\lceil m_j\rceil, \mathbf{v}\mathbf{A}_j - v_0\lfloor m_j\rfloor\,\} \tag{3.29}$$

where

$$m_j = \frac{\mathbf{v}\mathbf{A}_j - \mathbf{u}\mathbf{A}_j}{u_0 + v_0}$$

Note that this strengthening can be applied to more general disjunctions (i.e. not necessarily two term disjunctions, or involving just 0-1 variables). However, in that case, the expression for m_j is no longer a closed form, but has to be computed procedurally, in $O(p)$, where p is the number of terms in the disjunction. For details, see BALAS AND JEROSLOW[16].

Lift and project cuts without lifting and projection

Since the size of the cut generating linear program can be quite large relative to the size of the original disjuncts, solving it may prove too expensive for practical purposes. In fact, the experiments of BALAS ET AL[15] showed that the most efficient way to generate cuts using (CGLP$_k$) is to stop short of solving it to optimality. This was achieved by ignoring the multipliers (variables of (CGLP$_k$)) that correspond to constraints of $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ not tight at the optimum $\overline{\mathbf{x}}$ (except for the lower and upper bounding constraints on the 0-1 variables). The cuts were then lifted to the full $\mathbf{x}$ space using the closed form expression (3.22).

However, even when cuts are generated in a subspace as described above, this method still remains computationally challenging, due to the large size of (CGLP$_k$). Recent work by BALAS AND PERREGAARD[8] showed that it is in fact possible to compute the coefficients of α and β without actually solving the linear program (CGLP$_k$). The implication of this result is that we no longer need to *lift* the problem, by formulating it as a linear program, into the much higher dimensional space of $(\mathbf{u}, \mathbf{v}, u, v, \alpha, \beta) \in \mathbf{R}^m \times \mathbf{R}^m \times \mathbf{R} \times \mathbf{R} \times \mathbf{R}^n \times \mathbf{R}^n$ and then *project* it back onto the $\mathbf{x} \in \mathbf{R}^n$ space, but rather obtain the same inequalities directly in the $\mathbf{x}$ space. By performing certain pivots in the original simplex tableau, we can essentially simulate one or more pivots in the higher dimensional (CGLP$_k$). In other

words, we can determine the reduced costs of the dual multipliers $\mathbf{u}$ and $\mathbf{v}$ in (CGLP$_k$) by simply looking at a closed formula based on the current simplex tableau in the $\mathbf{x}$ space. Therefore, we can simply start with the original (optimal) LP solution $\overline{\mathbf{x}}$ and perform a sequence of pivots while monitoring what happens to these reduced costs. As long as there is at least one negative reduced cost, this means that we could further improve the objective function $\min \alpha\overline{\mathbf{x}} - \beta$ of (CGLP$_k$), which in turn means that we can improve the distance by which the inequality cuts off $\overline{\mathbf{x}}$. Due to this correspondence, the cut obtained in the end is exactly the cut we would have obtained by actually doing the pivoting in (i.e. solving to optimality) the larger (CGLP$_k$). This procedure is illustrated in Figure 3.1. More detailed steps of the algorithm can be found in Section A.1.

```
Algorithm Lift and project (k)

1   optimal ← false          /* will be set to true when we have an optimal lift-and-project cut from row k */
2   while (not optimal) do
        /* Are there negative reduced costs corresponding to the dual multipliers in CGLP_k? */
3       if (i* ← find_cut_improving_pivot_row (k) ) then
4           j* ← find_best_pivot_column (k, i*)
5           change_basis (i*, j*)
6       else
7           optimal ← true                                   /* we are done, the cut is optimal */
8       end if
9   end while

10  (α, β) ← create_intersection_cut (Ã, k)                 /* (α, β) is now a lift-and-project cut */
11  strengthen_cut (Ã, k, α)                                /* apply integrality strengthening */

12  return (α, β)
```

Figure 3.1: Generating lift-and-project cuts in the lower dimensional space

Two term simple disjunctions

Let us start our discussion of this algorithm by looking at the simple disjunction which is the one used to derive lift and project cuts, namely $(x_k \leq 0 \vee x_k \geq 1)$. This belongs to a very useful special class of disjunctions, namely two term simple disjunctions. They have the general form:

$$\mathbf{ax} \leq a_0 \vee \mathbf{bx} \geq b_0 \tag{3.30}$$

where $\mathbf{a} \in \mathbf{R}^n$, $\mathbf{b} \in \mathbf{R}^n$ and $\mathbf{x} \in \mathbf{R}^n_+$. When the two hyperplanes are parallel (that is, $\mathbf{a} = \mathbf{b}$ and $b_0 = a_0 + \epsilon$, where $\epsilon > 0$), these disjunctions are called *split disjunctions*. An important special case of split disjunctions is when $\mathbf{a}$, $\mathbf{b}$, a_0 and b_0 are integer and $\epsilon = 1$. This case is particularly interesting because there are no integer solutions between the hyperplanes, so by applying them to an integer program we lose no feasible points.

Split disjunctions and branching

Branching on an integer constrained variable x_k, whose value $\tilde{x}_k$ in the solution of the LP relaxation $P = \left\{ \mathbf{x} \in \mathbf{R}^n_+ : \mathbf{A}\mathbf{x} \geq \mathbf{b} \right\}$ is fractional, is equivalent to adding the split disjunction:

$$x_k \leq \lfloor \tilde{x}_k \rfloor \vee x_k \geq \lfloor \tilde{x}_k \rfloor + 1 \tag{3.31}$$

to the original formulation of the problem. In a typical branch and bound algorithm, however, the disjunction cannot be added explicitly, and it is instead enforced implicitly by creating two branches to be explored independently, one for each term of the disjunction. Since the language of integer programming does not include the $\vee$ logical connective, if we want to take advantage of its semantics (other than enforcing it through branching), we have to derive an implication of this disjunction that takes the form of a linear inequality.

Note that (3.31) does not exclude any integer solution that is feasible for the original problem. Indeed, its job is limited to preventing the fractional value $x_k = \tilde{x}_k$, and in fact any other fractional value $\lfloor \tilde{x}_k \rfloor < x_k < \lfloor \tilde{x}_k \rfloor + 1$, from ever being considered again. This means that any implication we derive directly from this disjunction will be valid for the convex hull of integer points that satisfy the original problem.

Intersection cuts from the current simplex tableau

A good point to start is the current LP solution, which is readily available in the simplex tableau. Let us extract the expression of x_k from the simplex tableau corresponding to the current solution $\tilde{x}$

$$x_k = \tilde{a}_{k0} - \sum_{j \in J} \tilde{a}_{kj} s_j \tag{3.32}$$

and plug it in (3.31). We obtain the following equivalent disjunction, which enforces

$\tilde{a}_{k0} - \sum_{j \in J} \tilde{a}_{kj} s_j \in \mathbf{Z}$:

$$\tilde{a}_{k0} - \sum_{j \in J} \tilde{a}_{kj} s_j \leq \lfloor \tilde{x}_k \rfloor \bigvee \tilde{a}_{k0} - \sum_{j \in J} \tilde{a}_{kj} s_j \geq \lfloor \tilde{x}_k \rfloor + 1 \tag{3.33}$$

We can now derive an inequality which is implied by this disjunction, by first changing the sign of the first disjunct, replacing $\tilde{a}_{k0} - \lfloor \tilde{x}_k \rfloor$ with f_{k0} to simplify the notation, and then using the inference rule (3.7):

$$\begin{array}{lcl} \sum_{j \in J} \tilde{a}_{kj} s_j & \geq & f_{k0} \\ \sum_{j \in J} -\tilde{a}_{kj} s_j & \geq & 1 - f_{k0} \\ \hline \sum_{j \in J} |\tilde{a}_{kj}| s_j & \geq & \min\{f_{k0}, 1 - f_{k0}\} \end{array} \tag{3.34}$$

Note that the current LP solution $\tilde{x}$ indeed violates this inequality, because the nonbasic variables s_j are all zero, while $0 < f_{k0} < 1$ (because $\tilde{a}_{k0} = \tilde{x}_k$ is the value of x_k in the current solution and $\tilde{x}_k$ is fractional). Since we were careful to construct (3.34) such that it remained valid for the convex hull of integer feasible points, we have essentially obtained a *cut* (a hyperplane that separates the current infeasible solution $\tilde{x}$ from the set of feasible points). Because we derived it by intersecting the disjunction (3.31) with the polyhedral cone obtained by relaxing P (i.e. dropping all the nonbasic inequalities), this cut is called an *intersection cut* (see BALAS[9]), and its coefficients can be computed in $O(n)$ from the kth row of the simplex tableau:

$$\begin{cases} \alpha_j = |\tilde{a}_{kj}| & \forall j \in J \\ \beta = \min\{f_{k0}, 1 - f_{k0}\} \end{cases} \tag{3.35}$$

Since $f_{k0} > 0$, we can use Remark 3.1.7 to normalize the disjunction before taking the cut in (3.34), and we obtain the following (generally) stronger cut:

$$\begin{cases} \alpha_j = \max\left\{ \frac{\tilde{a}_{kj}}{f_{k0}}, \frac{-\tilde{a}_{kj}}{1 - f_{k0}} \right\} & \forall j \in J \\ \beta = 1 \end{cases} \tag{3.36}$$

Note that this is not necessarily the only (or the best) normalization. For example, one that could be (computationally) more advantageous is obtained by multiplying the first

inequality with $1 - f_{k0}$ and the second with f_{k0}. We get:

$$\begin{cases} \alpha_j & = & \max\{\, \tilde{a}_{kj}(1 - f_{k0}), -\tilde{a}_{kj} f_{k0} \,\} \quad \forall j \in J \\ \beta & = & f_{k0}(1 - f_{k0}) \end{cases} \tag{3.37}$$

Expressing the cut in terms of the original variables

There is one more step we need to complete in order to be able to add this cut to the original model as an inferred inequality. We need to make sure that we have the cut expressed in terms of the original model variables. As it is now, $\alpha s_J \geq \beta$, we have no guarantee that the variables s_j appearing in the expression of (3.34) are all part of the original problem description (they could also be slack or surplus variables added by the simplex algorithm when it relaxed the problem to eliminate the $\leq$ and $\geq$ relational operators and transform it into a system of equations). So we now need to run Gaussian elimination again, this time to eliminate these extra variables (i.e. express them in terms of the originals). To perform this substitution of variables, note that originally, for each inequality $ax \leq b$ ($ax \geq b$), the simplex algorithm added a slack (surplus) variable s to transform it into the equality $ax + s = b$ ($ax - s = b$). So, for those nonbasic variables s_j that are not part of the original problem, we can simply replace s_j with the equation of the original row to which they were added as slack (surplus). That is, $s_j = b^i - a^i x$ ($s_j = a^i x - b^i$) for some row i. This way we obtain a cut $\alpha' x \geq \beta'$ expressed solely in terms of the original variables and we can safely add it to the model as an inequality implied by the original ones (*and*, of course, by the decisions we have made so far).

Special case: cuts from disjunctions on binary variables

Binary variables are a very useful modeling device in integer programming. For such variables $x_k \in \{\,0, 1\,\}$, we have that $\lfloor \tilde{x}_k \rfloor = 0$ in (3.34), so the intersection cut obtained by applying the disjunction $x_k \leq 0 \vee x_k \geq 1$ has the same coefficients as (3.35), but a simpler right hand side: $\beta = \min\{\tilde{a}_{k0}, 1 - \tilde{a}_{k0}\}$. When derived at the end of the pivoting procedure shown in Figure 3.1, this cut is a deepest *lift and project* cut (as shown by BALAS AND PERREGAARD[8]).

Strengthened lift and project cuts

The precise correspondence between the bases of the lower dimensional simplex tableau and those of the corresponding CGLP tableau enables us to find deepest lift and project cutting planes without going through the trouble of explicitly setting up and solving (CGLP$_k$) for each binary variable x_k which has a fractional value in the optimal LP solution. When and optimal solution to (CGLP$_k$) is found, through either procedure, we know that the cut can no longer be improved by finding better multipliers $\mathbf{u}$ and $\mathbf{v}$ (in other words, by adding multiples of inequalities in $\mathbf{Ax} \geq \mathbf{b}$ to either term of the simple disjunction $x_k \leq \lfloor \tilde{x}_k \rfloor \vee x_k \geq \lfloor \tilde{x}_k \rfloor + 1$). However, a further strengthening is possible by using integrality conditions on the remaining variables, as shown in Section 3.1.3. This, in fact, is the way to obtain mixed integer Gomory cuts from the optimal simplex tableau: we first derive an intersection cut from the tableau, without performing any additional pivots, as shown in (3.35) (or the stronger version (3.36)) and then apply integrality strengthening to it. On the other hand, the procedure outlined by BALAS AND PERREGAARD[8] for finding lift and project cuts, performs first a *sequence* of pivots and at the very end applies integrality strengthening to the resulting cut. This leaves open the following question: *would it not be possible to obtain a better cut by mixing the pivots with integrality strengthening steps*? This is the question we address in what follows.

Disjunctive modularization

BALAS[13] calls the strengthening procedure (3.29) *disjunctive modularization*. In order to implement it, we can modify the procedure of BALAS AND PERREGAARD[8] in the following way. Instead of pivoting all the way to optimality of (CGLP$_k$), and then applying disjunctive modularization to the resulting cut, we will apply it to the row of the simplex tableau corresponding to x_k *after each pivot*. This way, we effectively combine the two cut strengthening methods: *surrogates* (represented by pivots) and *integrality* (represented by disjunctive modularization (3.29)).

Cut generation procedure

The idea of the procedure is the following. We start from the optimal solution $\overline{\mathbf{x}}$ of

the LP relaxation (i.e. a vertex of the relaxed feasible region), derive an intersection cut from each fractional component $\overline{x}_k$ of $\overline{\mathbf{x}}$ and try to improve that cut by pivoting on the original LP tableau, using rules for pivot selection derived from the properties of (CGLP$_k$). The original procedure of BALAS AND PERREGAARD[8] generates lift and project cuts by first applying several rounds of strengthening by *combination* (i.e. pivots) and a final strengthening by *integer rounding*. We now use a modified version in which we apply the two strengthening methods in an interleaved manner: we apply integer rounding immediately after each pivot. Moreover, the intermediate (strengthened) cut is added to the model, which means that subsequent pivots (i.e. strengthening by combination) will take it into account. The process is repeated as long as there are negative reduced costs for the variables $\mathbf{u}$ and $\mathbf{v}$ of (CGLP$_k$).

Note that our goal is not to maintain feasibility and improve the objective function of the LP (which is essentially the goal of pivoting in the traditional simplex method), but to obtain a better cut, by stepping around (and possibly outside) the feasible region of LP. Therefore, our pivots will quite likely take us to points in space where one or more constraints of the original LP don't hold. For this reason, we need to save the optimal basis of the LP before starting generating cuts and restore it when we are done with cut generation, so that the integer programming solver does not get confused when trying to make search decisions.

Experimental results

We ran experiments the evaluate the effectiveness of interleaving the two cut strengthening methods. For these experiments, we used the latest version of the MIPLIB library of benchmark problems, MIPLIB 2003. Table A.1 shows the characteristics of the problems used: the number of rows and columns in $\mathbf{A}$, the number of binary components and the number of integer (non-binary) components of $\mathbf{x}$. All results reported here were obtained using XPRESS-MP version 15.3. In order to evaluate the impact of the resulting lift and project cuts in the cleanest possible way, we turned off the presolve and cut generation functions of XPRESS. For details on both the algorithm and on the results of these experiments, please see Appendix A.1.

3.2 Constraint programming

The theory of *constraint programming* (CP) represents a significant departure from the philosophy of its predecessors. We are no longer restricted to a closed language, as it was the case with linear, mixed integer linear and disjunctive programming, but we can now freely define new operators (constraints), with their own semantics and possibly new inference rules. The most obvious such operator would be $\neq$, which now allows us for the first time to reason about *linear disequations* (if you remember our discussion of linear and mixed integer linear programming, such reasoning is not possible within the framework of those theories). There are already several hundred such operators, some more frequently encountered than others. The interested reader can find more information about these operators (*global constraints* in the CP community) in the catalog maintained by BELDICEANU[20].

3.2.1 Modeling and solution process

The fact that constraint programming allows a higher degree of modeling flexibility makes it an attractive alternative to linear and integer programming. However, before a new operator can can be added to the language, it must be equipped with specialized inference algorithms in order to be useful in deriving proofs. Such inference algorithms are commonly referred to as *filtering algorithms*, or *domain reduction algorithms* in the CP community. For many operators, there may exist multiple such algorithms, with varying complexity and performance guarantees. For instance, some algorithms guarantee completeness (that is, after they are applied, no values are left in the domains of their variables which do not belong to some feasible solution). This property is known as *hyper-arc consistency*. Others do not provide such guarantees, but execute much faster, and are therefore preferable under certain circumstances.

Constraint solvers employ a seemingly different approach than integer solvers, in the sense that at every step of the process, each constraint is responsible for drawing inference in isolation. In integer programming, inference is typically obtained from a collection of constraints and from the objective function, considered together. When no more inference

is possible, the search process is invoked. This time, the process is similar to the one used in integer programming: they both share a divide and conquer type of strategy. The CP community, however, was far more active in this area than the IP community, and so there are many more variants of divide and conquer search encountered in CP than in IP, such as limited discrepancy search (see HARVEY AND GINGSBERG[75]) and first-fail search (see HARALICK AND ELLIOTT[42]).

One important drawback of constraint programming, when compared with integer programming, is that it cannot generate useful bounds on the value of an objective function, if one is present. In integer programming, the simplicity of constraints allows the solver to automatically generate a convex reformulation of the problem (by linearizing the integrality constraints), which can then be used to compute a bound on the value of the objective. CP constraints do not offer this flexibility, and this is the main reason why CP-based methods have not been as successful for optimization problems as IP-based methods. However, when dealing with pure feasibility questions, CP solvers are typically much more effective than IP solvers, since they have access to more information about the nature of problem (and therefore its solutions) than pure IP models can provide.

This contrast of capabilities, together with the fact that parts of their solution processes overlap, makes the combination of the two techniques desirable. This is the starting point of our discussion of a unifying framework, in which these two (and other) methods can work together, at each step of the solution process, in order to take advantage of their complementary and common strenghts.

Chapter 4

A unifying framework

The idea of a unifying framework for optimization has been around for over a decade, but very few attempts have been made at providing a description of its major components and their interaction. A notable exception is the recent book of HOOKER[50], which builds on some of the earlier work of CHANDRU AND HOOKER[28], HOOKER[48] and ARON, HOOKER AND YUNES[6] to provide a consistent view of optimization techniques as they are used in mathematical and constraint logic programming. The framework we describe here has evolved from a joint work with HOOKER AND YUNES[6], and it provides the basic blocks for a modeling language of metaconstraints, as well as the mechanisms that a generic metasolver needs in order to solve a problem according to the specification given by the metaconstraints. The solution process employed by this framework is an interleaving of two fundamental steps, search and inference, each of which is reflected *at the modeling level* in the language of metaconstraints. Inference can be drawn using multiple techniques from (a) individual constraints (similar to domain reduction in constraint programming or bound preprocessing in integer programming) or (b) groups of constraints (similar to relaxation solving in mathematical programming). Search is performed using constraint selection rules and search direction selection rules, which can be statically specified by the modeler and dynamically prioritized by the inference component during the solution process.

4.1 Modeling

A key decision in optimization is the choice of a modeling language. Modeling must be done at a high enough level so that the problem statement appears clear to the modeler and at the same time provides sufficient structural information to the underlying solver. We saw earlier that in order to efficiently prove that the pigeonhole problem does not admit solutions if there are more pigeons than holes, we needed to state the problem in a language whose semantics offered powerful enough tools for such inference. We needed to communicate to the solver that it should use *counting arguments*. In this case, we reformulated the problem from a statement in propositional logic to one in 0-1 integer programming, and later to a continuous linear programming problem. It happened that in this case the translation was straightforward. The difficulty, however, is that we cannot efficiently represent any arbitrary problem in propositional logic. So we need a more expressive language, but which still allows us to easily reformulate problems into perhaps continuous linear programming, or even other paradigms.

Representability

One language which could easily provide such immmediate reformulation rules is the language of mixed integer programming. However, mixed integer programming itself is not expressive enough to allow one to efficiently state all the problems one might need to solve. In fact, JEROSLOW[55] showed that the expressiveness of mixed integer programming is surpisingly limited, by proving that a set $S \subset \mathcal{R}^n$ is mixed integer representable if and only if it is a finite union of polyhedra whose set of recession directions coincide. In other words, even simple statements such as "x is different from 3" $(x \neq 3)$ have no representation in mixed integer programming. Constraint programming, on the other hand, provides an open language, in which problem characteristics can be encoded into new constraints, as necessary. Thus, an unrestricted combination of theories can be used when deriving inference, which is a very desirable property. This is the model we want for a language of *metaconstraints*, since it gives us the flexibility to add specialized inference algorithms to these constraints, or create new constraints when the existing ones are not expressive enough.

4.1.1 Metaconstraints

Informally speaking, a metaconstraint is a relation. It operates on the variables of a problem and produces a true or false answer depending on whether there exists a replacement of the variables with constants which turns the relation into a true statement. A metaconstraint can itself be viewed as an *independent* problem and considered in isolation. This is an important observation, because it tells us that even if we have a very complex problem, consisting of a large set of metaconstraints, proving that the problem represents a false statement (that is, it has no solution) could be as easy as proving that any one of these metaconstraints, or subproblems, is false (infeasible). This in fact is the approach taken in constraint programming. The solution process there is based solely on this type of infeasibility proof: the *specialized inference* algorithm(s) of each constraint are queried in isolation, and if one of them responds with false, the problem itself is false. What constraint programming notably lacks is the second type of inference, which is mostly used in mathematical programming, namely *inference via reformulation*. Besides providing additional tools for inference which are not available in the original language, reformulation has the great advantage that, if constraints are reformulated into a *common* language, then we could draw inference based on *all* at once, rather than querying each of them separately, as it is the case in constraint programming. This way we obtain a more *global view* of the problem, which may not necessarily be available in the original language. This global view is actually what's behind HOOKER[50]'s choice to distinguish between inference and relaxation, since the relaxation is precisely a *collection* of constraints that have been reformulated into the same language.

Multiple reformulations

It should be clear by now that it is useful to equip metaconstraints with *translation rules*, preferably rules that translate into some common language. In fact, not only can we associate with a metaconstraint one set of translation rules for each given target language, but we can create alternative translations of that metaconstraint into the same language. To see why this is advantageous, let us examine the following example.

Suppose that we want to reformulate the following metaconstraint

$$\sum_{i=1}^{n} |x_i| \leq 1 \qquad (C_1)$$

into an integer program. As it is stated, this constraint cannot be used in an integer programming solver, because the "absolute value" ($|x|$) is not part of the language of integer programming. One plausible reformulation would be to enumerate all possible cases of signs. For example, for $n = 2$, the constraint

$$|x_1| + |x_2| \leq 1$$

would be reformulated as the system of four inequalities

$$\begin{aligned} -x_1 \quad -x_2 &\leq 1 \\ -x_1 \quad +x_2 &\leq 1 \\ x_1 \quad -x_2 &\leq 1 \\ x_1 \quad +x_2 &\leq 1 \end{aligned} \qquad (C_1 \rightarrow R_1)$$

representing all possible cases. Clearly, this is not a desirable representation for practical purposes, as it requires all the 2^n inequalities in the general case (see JEROSLOW[54]).

However, an alternative reformulation is possible, with a much more compact representation. If we define a new variable $y_i \geq 0$, then we can write a reformulation of (C_1) which is linear in size:

$$\begin{aligned} \sum_{i=1}^{n} y_i &\leq 1 \\ x_i &\leq y_i \quad \forall i \\ -x_i &\leq y_i \quad \forall i \end{aligned} \qquad (C_1 \rightarrow R_2)$$

Note that this new model is much smaller in size than ($C_1 \rightarrow R_1$), even though it actually requires more variables.[1]

Constraint syntax

We can now define a new metaconstraint based on (C_1) which knows how to reformulate itself in the language of integer programming (and therefore automatically into linear programming).

[1] So it is not always a good idea to stick to models with fewer variables!

```
C1 {
    sum i of | x[i] | <= 1
    infer using C1 → R2
}
```

Figure 4.1: Modeling: example of a metaconstraint with reformulation rules

In doing this, we implicitly assume that the implication $C_1 \rightarrow R_2$ is known to the language, via a keyword which links to the special translation procedure outlined above for the reformulation of the metaconstraint (C_1) into $(C_1 \rightarrow R_2)$. This procedure is stored in a module, which does not need to be seen or used directly by the modeler. Upon reading the model, the metasolver brings in the module and uses it to obtain the system described by $(C_1 \rightarrow R_2)$.

Note that there is nothing to prevent us from reformulating the constraint in multiple ways. For instance, we could declare it as in Figure 4.2. However, this does not make

```
C1 {
    sum i of | x[i] | <= 1
    infer using C1 → R1, C1 → R2
}
```

Figure 4.2: Modeling: metaconstraint with multiple reformulation rules

much sense and would not be useful at all in this case, for two reasons. One, we saw that the first reformulation we came up with, namely $(C_1 \rightarrow R_1)$, had an exponential number of constraints. Secondly, even if it had a more reasonable size, it would still not be of much use since both $(C_1 \rightarrow R_1)$ and $(C_1 \rightarrow R_2)$ translate the constraint into the same language, which is somewhat redundant. The idea is to be able to obtain reformulations in a variety of languages, and not multiple reformulations into the same language. However, of course, having more than one reformulation into a given language allows us to choose the one better suited to the problem at hand.

4.2 Solution process

We have argued so far that the solution process in optimization is a combination of two fundamental steps: search and inference. These two steps are interleaved and rely on each other in order to advance towards a solution of the problem or prove that none exists. The advancing process is done by the search component, while the verification of feasibility is handled by the inference part. Figure 4.3 gives an outline of this solution process.

```
Algorithm solve (P)

1   I ← ∅                                  /* I is the set of inferences made by the inference component */
2   D ← ∅                                  /* D is the set of decions made by the search component */
3   while (∃ unexplored ⟨variable, constant⟩ substitutions) do
4       I ← infer (P, I, D)
5       if (false) then
            /* The inference component decided that we made some wrong decisions along the way, so we must backtrack */
6           (D, I) ← backtrack (P, I, D)
7       else
8           if (P is TRUE) then
                /* All variables have been eliminated and we have determined that P is true (consistent) */
9               return true
10          else
                /* We are not done yet, but no further inference is possible, so we'll have to make some new decisions */
11              D ← search (P, I, D)
12          end if
13      end if
14  end while
15  return false                           /* All possible elimnations were tried and none turns P into a true statement */
```

Figure 4.3: General algorithm to solve an optimization problem P

The algorithm starts with the statement of P and searches over all possible ways to eliminate variables from P. The search is kept in check by the inference module, which is called at every stage in order to (a) detect wrong decisions made earlier, and (b) help the search avoid new ones. The state of the algorithm is kept in two sets, $\mathcal{I}$ and $\mathcal{D}$. The set $\mathcal{I}$ contains all the inferences made by the inference module up to this stage. Similarly, $\mathcal{D}$ is the set of all decisions made by the search module up to now. Note that new elements are added to $\mathcal{I}$ only by the inference module, while $\mathcal{D}$ is augmented only by the search

module. However, whenever a dead end is detected during inference (that is, the inference module decides that the combination of the original statement P with $\mathcal{I}$ and $\mathcal{D}$ leads to a false statement), both sets are modifed by the *backtrack* module. This module has the role of reversing some of the decisions made by the search module, and with them the inferences from $\mathcal{I}$ that were made *on the basis* of these decisions. The algorithm ends when all possible decisions have been either explored by the search or ruled out by the inference module. The decision that P is true is made when at least one elimination was found which leads to a true statement and either we exhausted all other eliminations or inference tells us that there are no others which could provide a better conclusion (in the case of optimization, no other eliminations will provide a solution with a better value).

Note that there is no reference to an objective function. This is because the only role of the objective function in optimization is to help the inference process, by using *bounding arguments*, and so all references to such a function are captured inside the inference module.

4.2.1 Search

Search becomes useful when inference alone cannot decide on the feasibility or infeasibility of the problem. In general, search takes the form of variable elimination, in the sense that it attempts to remove variables from the problem statement by replacing them with constants from their corresponding domains. However, other ways to conduct search are possible, and it may be the case than sometimes a better decision is not to eliminate a variable, but instead to just restrict the number of possible replacements we can make for that variable. This is because sometimes inference is able to determine infeasibility even if the variable is still in the problem, based only on its restricted domain.

The typical search process is mostly unguided. Inference may try to suggest ways for the search to proceed, but this is in general not well understood and poorly used in most existing solvers. In integer programming, the way search proceeds in most solvers is by looking at the solution of the relaxation, which in general has little or nothing to do with what we are in fact looking for. There are examples in which such a solution is arbitrarily far from any feasible integer solutions, and using it as a guide for the search is

indeed a very poor decision. The integer solver will attempt to search so that it corrects a problem it sees in the solution of the relaxation. It does so by choosing some variable x_k which does not have an integer value in the relaxation and creating a disjunction of the type $x_k \leq \lfloor \tilde{x_k} \rfloor \vee x_k \geq \lceil \tilde{x_k} \rceil$, where $\tilde{x_k}$ is the value of x_k in the solution found by the continuous solver. This is a very weak way of searching, since there is no guarantee that either (1) x_k will receive an integer value, or (b) any of the variables which happened to be integral in the current relaxation solution will remain so.

Search rules

A better way to search is to exploit knowledge about the problem itself, and the best handle we have into the problem structure is the collection of metaconstraints. Let us illustrate this with a simple example (which happens to be useful also in the context of integer solvers). Consider the linear constraint

$$x_1 + x_2 + x_3 \leq 1 \tag{C_2}$$

where $x_i \in \{0, 1\}, \forall i$ and suppose that one of the variables was assigned a fractional (non-integer) solution by the continuous solver during the inference process (remember that the solver attempts to bound the current problem using the objective function, and in the process it finds the most extreme point of the polyhedron - that point is the solution we are using here). One option is to create two search directions (branches), using the disjunction we mentioned above, in order to "avoid" this particular fractional solution in the future. A wiser choice would be to avoid *all* fractional solutions in the future, not just this one. We can do so by creating four search directions, each of which eliminates all three variables at once:

$$\begin{aligned} &\text{direction 1:} \quad x_1 \leftarrow 0,\ x_2 \leftarrow 0,\ x_3 \leftarrow 0 \\ &\text{direction 2:} \quad x_1 \leftarrow 1,\ x_2 \leftarrow 0,\ x_3 \leftarrow 0 \\ &\text{direction 3:} \quad x_1 \leftarrow 0,\ x_2 \leftarrow 1,\ x_3 \leftarrow 0 \\ &\text{direction 4:} \quad x_1 \leftarrow 0,\ x_2 \leftarrow 0,\ x_3 \leftarrow 1 \end{aligned} \tag{$C_2 :: S_1$}$$

This way of searching covers all feasible cases, and avoids potential fractional (and therefore infeasible) ones. It could therefore lead to a much shorter proof, although this generally depends, of course, on the rest of the problem.

```
C2 {
    sum i of x[i] <= 1
    search using C2 :: S1
}
```

Figure 4.4: Modeling: example of a metaconstraint with search rules

So we could also equip this metaconstraint with a *search rule*, which we can later refer to when we use it in the statement of a problem. The resulting metaconstraint is shown in Figure 4.4. Just as it was the case with inference rules, the language is now aware of the keyword $C_2 :: S_1$, and it uses it to make search decisions based on this metaconstraint. Note that this way of specifying search rules for individual constraints does not prevent us from using the already existing strategy from integer programming, namely adding the disjunction $x_k \leq \lfloor \tilde{x_k} \rfloor \vee x_k \geq \lceil \tilde{x_k} \rceil$, when the value $\tilde{x_k}$ of x_k is fractional in the continuous linear relaxation. Indeed, we can use a metaconstraint called **integer(x_k)**, whose search rule produces:

$$\begin{aligned} &\text{direction 1:} \quad x_k \leq \lfloor \tilde{x_k} \rfloor \\ &\text{direction 2:} \quad x_k \geq \lceil \tilde{x_k} \rceil \end{aligned} \qquad (int :: frac)$$

The metaconstraint could then be declared as in Figure 4.5.

```
integer {
    x is int
    search using int::frac
}
```

Figure 4.5: Modeling: example of a metaconstraint enforcing integrality of variables

Rule selection

Since a model is a collection of metaconstraints, and each metaconstraint has its own search rule, we must be able to decide which rule to use at a given point during the search. Obviously we can only use one, but which one we choose can have dramatic effects on the length of the search. The rule selection is performed dynamically by the underlying

metasolver. As pointed out earlier, the main task of inference is to come come up with a proof of infeasibility. If it fails, there are still some positive side effects of the inference process, such as *reduced variable domains*, *cutting planes*, *new bounds on the objective function*, and so on. This additional information can be used in order to come up with recommendations for the search. One possible strategy, which is also the one we employ in our implementation (see ARON ET AL[6]), is to evaluate how close to infeasibility each metaconstraint is (given the new information) and then select the one which is closest to infeasibility (or feasibility) in order to decide the next search direction. Once such a constraint is selected, the particular search decisions, and their order, are left to its search rule. In order to allow the modeler some flexibility in choosing what the inference module

```
search {
    create  using  least::{C2, C3}, most::{C1, C4}, first::all
    explore using  best::bound::dive
}
```

Figure 4.6: Modeling: example of search module specification

should recommend, one must either allow procedural elements into the language, or at least provide some declarative means of choosing among the most common alternatives. In ARON ET AL[6], we opt for the latter, since the goal is to provide a framework that is powerful enough while still easy to grasp and use. The statement in Figure 4.6 says that search should first consider the rules of constraints C_2 and C_3, and use the search rule of the *least* feasible of the two. If the two are feasible, then search will consider the next group of constraints, C_1 and C_4, and use the one which is closest to feasibility, as indicated by the keyword *most* (feasible). If both these constraints are feasible as well, then it selects the first infeasible (*first*) from the remaining constraints (*all*) to continue the search. If no infeasible constraint exists, a solution (and perhaps a new incumbent) has been found, so the search can now focus on some unexplored direction, as discussed next.

Direction selection

As it is clear from the previous examples, each search rule will generally add more than one direction to the search. Since only one such direction can be explored at a time, the rest are kept in a pool. This pool can be prioritized in various ways, which are indicated by keywords like *best::bound::dive*, or *best::bound::explore* (see Figure 4.6) representing commonly used pool management strategies. The search module always picks the direction which is at the top of this pool. Internally, the pool is stored as a multiple key priority queue, to allow ordering based on several characteristics, including search depth (number of decisions made from the original problem statement), objective bound, degree of infeasibility, etc. The main search procedure takes a very simple form, as shown in Figure 4.7, since most of the search related decisions are left to the constraints. This is the procedure being called in line 11 of Figure 4.3.

Algorithm **search** $(P, \mathcal{I}, \mathcal{D})$

```
   /* Find the one constraint which the inference engine has ranked first as most promising for search */
1  foreach (constraint group G_i of P) do
2      if (∃c ∈ G_i | c is infeasible according to I) then
3          select the first such constraint c and break
4      end if
5  end for

   /* Generate new search directions based on the search rule of the selected constraint */
6  D ← D ∪ c.search_rule ()

   /* D now reflects the current state of the search, plus the new alternatives generated by the search rule of c */
7  return D
```

Figure 4.7: Search algorithm based on metaconstraint search rules

4.2.2 Inference

The inference module is called upon before and after a search decision is made. The role of this module is to decide whether the current state of the search is infeasible. While attempting to prove infeasibility, the module can use a variety of tools and techniques, as instructed by the metaconstraints. It may happen, and it is often the case, that inference

cannot (yet) conclude that the problem is infeasible, so further search is required. Even when this is the case, important information becomes available as a result of attempting to prove infeasibility, which can be used to further limit the number of possibilities available to the search module.

Using multiple reformulations

As we have seen, each metaconstraint is equipped with special inference rules, all of which essentially reinterpret the constraint in some theory and attempt to prove its infeasibility using that theory. For instance, symbolic constraints such as *alldifferent(x)* can be reformulated as graph theoretical problems, and solved using theorems from *graph theory* (see RÉGIN[69]). Others have better translations into *continuous linear programming*, and many can even be reformulated efficiently in multiple languages. We could, for instance, draw inference from the *alldifferent(x)* constraint both by using the graph theoretical reformulation, as well as by reformulating it as a linear programming problem, although the linear reformulation is not nearly as useful in practice for this constraint. However, it is an instructive exercise.

The *alldifferent(x)* constraint states that the components of an n-dimensional vector of discrete variables $\mathbf{x} \in \{1, \ldots, m\}^n$ must be pairwise different. In other words, no two x_i and x_j with $i \neq j$ can have the same value. We could start a reformulation process by first stating the constraint using a simpler relation, "$\neq$" in place of the more complex "alldifferent". Thus, we can write

$$x_i \neq x_j, \forall 1 \leq i < j \leq n$$

which is semantically equivalent to the original constraint. Since the "$\neq$" relation is not part of the language of linear programming, we must replace it with inequalities. An equivalent statement involving inequalities would be

$$x_i \neq x_j \leftrightarrow x_i < x_j \vee x_i > x_j$$

but this is still not representable as a linear program (neither "$<$" nor "$\vee$" are part of that language). Fortunately, since x_i and x_j can only take discrete values, we can escape

the "$<$" relation and replace it with "$\leq$", which is allowed in linear programming:

$$x_i < x_j \vee x_i > x_j \leftrightarrow x_i \leq x_j - 1 \vee x_i \geq x_j + 1$$

We still need to eliminate the "either-or" disjunction relation "$\vee$". One way to do this is to introduce additional variables into the model, one for each disjunct, to play the role of selectors for which disjunct is true. Let $z_{ij} \in \{0, 1\}$ be such a variable to indicate which of the two inequalities $x_i \leq x_j - 1$ and $x_i \geq x_j + 1$ holds true. We can now write a linear system which is implied by the original constraint:

$$\begin{aligned} x_i - x_j + 1 &\leq m z_{ij} & \forall 1 \leq i < j \leq n \\ x_j - x_i + 1 &\leq m(1 - z_{ij}) & \forall 1 \leq i < j \leq n \end{aligned} \qquad (aldiff \rightarrow linear)$$

Whenever $z_{ij} = 0$, the system is equivalent to $x_i \leq x_j - 1$ (the second inequality becomes true since no two numbers in the set $\{1, \ldots, m\}$ can be m units apart), and when $z_{ij} = 1$, it is equivalent with $x_i \geq x_j + 1$ (the first inequality becomes true). Thus,

```
alldifferent {
    alldiff(x)
    infer using linear::default, specialized::default
}
```

Figure 4.8: Modeling: alldifferent with reformulation in multiple languages

we can declare a metaconstraint *alldifferent* using two different inference (reformulation) rules, as shown in Figure 4.8. The first reformulation (default linear) points to a procedure which creates $aldiff \rightarrow linear$. The second reformulation (specialized) of *alldifferent(x)*, into a graph theoretical problem, converts the statement of alldifferent into a *bipartite matching* problem. On one side of the graph are the variables x_i, on the other side their possible values. Then alldifferent is infeasible if there is no matching that covers all variables. To determine that no such matching exists, the algorithm proposed by Régin[69] attempts to remove all edges from the graph that are not part of a maximum cardinality matching, using a theorem due to Berge[21]. This algorithm is what is invoked by the framework upon encountering the `specialized::default` rule in the definition of the alldifferent metaconstraint in Figure 4.8.

There is an interesting distinction to be made here between the two reformulations of the *alldifferent(x)* metaconstraint, which in fact is behind HOOKER[50]'s view of relaxation and inference as being different. The graph theoretical reformulation, $aldiff \rightarrow regin$, is a *specialized* one, meaning that it was not intended to be used in combination with reformulations of other metaconstraints into graph theory (although it could, in theory, but nobody has looked in that direction so far). The other reformulation, into continuous linear programming, $aldiff \rightarrow linear$, can in fact be used *in conjunction with* reformulations of other metaconstraints as linear programs, in order to strengthen the inference by taking more aspects of the problem into account at the same time. Hooker's definiton of a relaxation is based precisely on this property of the continuous reformulation: namely that it can collect implications from more than one constraint and attempt to prove infeasibility based on *all* of them at once. In fact, the definition he uses starts from an alternative, equivalent viewpoint: it regards a relaxation as being obtained from the original problem by *dropping* some of the original constraints form the problem statement. Regardless of the viewpoint, there is a clear advantage in using relaxations (as *collections* of constraint implications), since in many cases it may be difficult to prove infeasibility by looking at constraints in isolation, but once two or more constraints are considered together, infeasibility becomes much easier to detect.

Relaxation as a special case

The framework proposed here includes the notion of relaxation as a special case, since it allows for reformulations of individual constraints to *also* be collected into a common implication whenever the target theory allows it. In particular, this is done automatically for continuous linear reformulations, as well as reformulations into propositional logic. For specialized reformulations such as Régin's graph theoretical view of *alldifferent*, inference is drawn only in isolation from each such reformulation at a time. In fact, this is also done for general reformulations: the method first attempts to prove that at least one of the individual reformulations is infeasible and *only if* no infeasibility is detected, the globalized reformulation (which could be quite large for some problems) is considered. The outline of the inference algorithm is given in Figure 4.10.

```
infer {
    linear  using  gomory::60 , lnp::10 , mir , flow
    logic   using  robinson
}
```

Figure 4.9: Modeling: example of global inference specification

Inference from relaxations

In order to take better advantage of relaxation as a special case of reformulation, we should instruct the underlying solver on what sort of inference it must draw from such relaxations. For instance, with a collection of linear constraints, we can derive cutting planes of various types and strengths, bounds, etc. For a collection of logical clauses, we can use resolution. The default behaviour is to use all the available techniques for a given theory, but this can be further specialized by adding another section to the model, as shown in Figure 4.9. This says that for the collection of linear implications of the problem (that is, the linear *relaxation*) inference should use cutting plane generators, and in particular *Gomory mixed integer* cuts (see GOMORY[38]), *lift-and-project* cuts (the algorithm described in Chapter 3.1.3), *mixed integer rounding* cuts (see MARCHAND ET AL[63]) and *flow cover* cuts. Furthermore, it should only keep the best 60% of the Gomory cuts and 10% of the lift and project cuts that are generated, ranked by the distance from the solution of the linear relaxation (this limitation ensures that subsequent inference steps will not take too long, which can happen when too many such cuts are being added to the formulation).

Guiding the search

The degree of feasibility of each constraint is determined by the inference module, which also provides a ranking of the constraints according to feasibility degrees. As shown in Figure 4.6, constraints can be grouped by priority, in which case the inference module ranks constraints in each group separately, to allow for even more specialized search decisions. Using this ranking mechanism, the search module can make a *deterministic* decision. This decision is based on as much information as we can obtain during the inference process and therefore reflects the problem structure as revealed by the metaconstraints.

Algorithm **infer** $(P, \mathcal{I}, \mathcal{D})$

> This first loop is similar to what constraint programming solvers do - run inference on each constraint in isolation. However, the procedure described here is more general, because it allows multiple reformulations of the same constraint to be used.

```
    /* Try each individual constraint reformulation and see if any comes back infeasile */
1   foreach (metaconstraint C_i ∈ P) do
2       foreach (reformulation rule C_i → R_i) do
3           if ( I ← infeasible (C_i → R_i, I) ) then
4               return false
5           end if
6       end for
7   end for
```

> The remaining part is similar to what mathematical programming solvers do - run inference on a relaxation of the entire problem. The procedure described here is more general, because it also allows multiple reformulations of the problem to be used.

```
    /* No individual constraint could be proved infeasible, so we construct problem reformulations */
8   foreach (reformulation rule C_i → R_i) do
9       if (theory T^{R_i} of R_i allows global inference) then
10          P^{T^{R_i}} ← P^{T^{R_i}} ∪ R_i          /* Add the reformulation R_i of C_i to relaxation P^{T^{R_i}} */
11      end if
12  end for

    /* Try each problem reformulation now and see if any comes back infeasile */
13  foreach (reformulation P^{T^{R_i}} of P) do
14      if ( I ← infeasible (P^{T^{R_i}}, I) ) then
15          return false
16      end if
17  end for

    /* No infeasibility detected, so we just return the inferences made along the way */
18  return I
```

Figure 4.10: Inference algorithm based on metaconstraint inference rules

4.3 Example: Production planning

Let us illustrate these concepts with a practical example, namely a production planning problem. The objective is to manufacture several products on a line of limited capacity C so as to maximize net income. Each product must be manufactured in one of several production scale modes (small, medium, large), or not manufactured at all, and only a certain range of production quantities are possible for each product in each mode. Thus if x_i units of product i are manufactured in mode k, $x_i \in [L_{ik}, U_{ik}]$. The income $f_i(x_i)$ obtained from making product i is linear in each interval $[L_{ik}, U_{ik}]$, which means that f_i is a *semicontinuous piecewise linear function* (Figure 4.11):

$$f_i(x_i) = \frac{1}{U_{ik} - L_{ik}} \left[(U_{ik} - x_i)c_{ik} + (x_i - L_{ik})d_{ik} \right], \quad \text{if } x_i \in [L_{ik}, U_{ik}] \tag{4.1}$$

The mode $k = 0$, for which $[L_{i0}, U_{i0}] = [0, 0]$, corresponds to the case when product i is not produced at all.

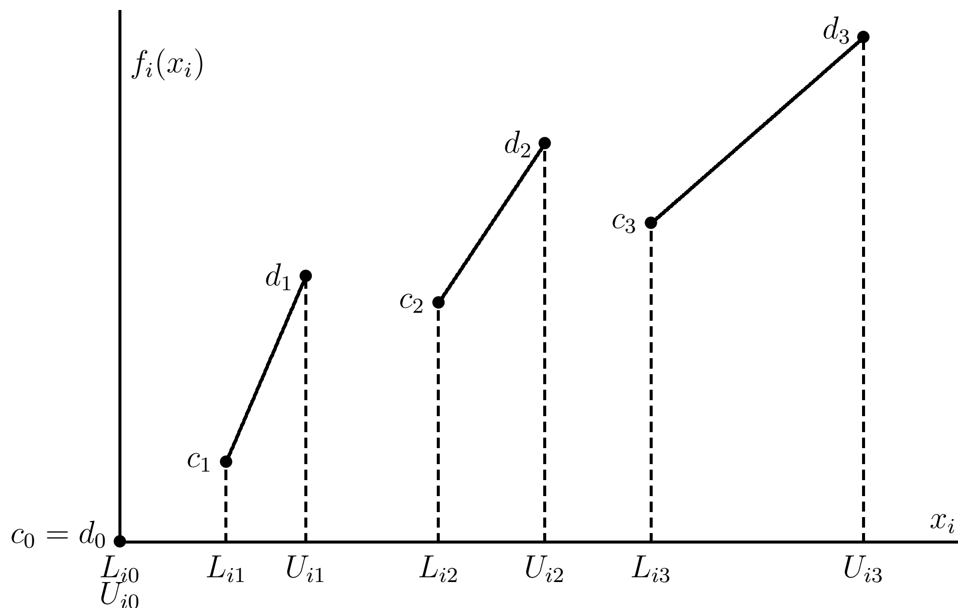


Figure 4.11: Production planning: a semicontinuous piecewise linear function $f_i(x_i)$.

4.3.1 Modeling

An integer programming approach to modeling this problem requires the introduction of some indicator (0-1) variables y_{ik} to select the production mode of each product. The functions f_i can be modeled by assigning weights λ_{ij}, μ_{ik} to the endpoints of each interval k, and so a linear model for this problem could look like the following:

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

CP model

However, a perhaps more natural way to model this problem is to use *conditional constraints* of the form $A \Rightarrow B$, which means that whenever the *antecedent* A is true, the *consequent* B must be enforced. Note that this is *not* the same as an implication, since it does not say that whenever B is not true, A must also be false. In this case, it simply acts as an on-off switch for constraint B based on the truth value of A. Furthermore, the piecewise linear cost function $f_i(x_i)$ can be coded much as it appears in (4.1), so the new model looks like:

$$\begin{aligned}
&\max \sum_i u_i \\
&\sum_i x_i \leq C && (a) \\
&x_i \in [L_{ik}, U_{ik}] \Rightarrow u_i = \tfrac{1}{U_{ik} - L_{ik}} \left[(U_{ik} - x_i) c_{ik} + (x_i - L_{ik}) d_{ik} \right], \text{ all } i, k && (b)
\end{aligned} \tag{4.3}$$

Note that no discrete variables are required, and the model is quite simple. This model can now 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 $[L_{ik}, U_{ik}]$.

A model based on metaconstraints

Although the above model is a perfectly legitimate approach to solving the problem and might appear as quite natural initially, it does not fully exploit the problem's structure. Note that for each product i, the point (x_i, u_i) must lie on one of the line segments defined by consequents of constraint (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 a (potentially) faster solution.* This can be accomplished by equipping the modeling language with a metaconstraint *piecewise* which models continuous or semicontinuous piecewise linear functions. A single piecewise constraint represents the constraints in (b) that correspond to a given i. The model now becomes

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

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 $u_i = f_i(x_i)$. The high level representation of this model, in compilable form, is shown in Figure 4.12. The key points in the model are the inference rules on the piecewise constraint (which is linearized using the default available algorithm and produces the convex hull (4.5) of feasible values of x_i and u_i *given the current state of the search*), the search rules that say we should branch on the least feasible piecewise constraint (and use the algorithm "three_split" in order to create new branches) and the branch selection rule which says that we should select the branch with the best bound and then dive from

there until a leaf is reached in the search tree. The degree of infeasibility of each piecewise constraint is calculated by measuring how far the values of x_i and u_i obtained from the linear solver are from the closest linear segment (piece) of the function.

```
maximize {
    income = sum i of u[i]
}

capaticy {
    sum i of x[i] <= C
}

piecewise {
    __piecewise(x[i],u[i],L[i],U[i],c[i],d[i]) forall i
    infer   using specialized::default, linear::default
    search  using three_split
}

infer {
     linear  using  gomory::40 , mir::15
}

search {
    create  using  least::{ piecewise }
    explore using  best::bound::dive
}
```

Figure 4.12: Production planning: integrated model

4.3.2 Solution process

Let us suppose the solver is instructed to solve the problem by *best::dive* (a variant of branch and bound), which is defined as a direction exploration strategy in the *search* component of the model in Figure 4.12. 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 (necessary for inference on the linear reformulation, as well as for ordering new restrictions in a pool as indicated *best::bound::dive*) by solving a linear relaxation of the problem. This

relaxation is (4.5). The solver branches whenever a constraint is violated by the solution of the relaxation. The nature of the branching is dictated by the constraint that is violated.

Specialized inference

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 $\min\left\{b_i, C - \sum_{j \neq i} a_j\right\}$. 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.

Inference via reformulation

The solver creates a linear relaxation at each node of the search tree by pooling linear reformulations of the various constraints. For example, it reformulates each constraint in (b) by generating linear inequalities to describe the convex hull of the graph of each f_i, as illustrated in Figure 4.13. 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 contributes to the relaxation, by simply sending a reformulation of itself (identity). These relaxations, along with the domains, combine to form a linear relaxation of the entire problem:

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

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

The solver next finds an optimal solution $(\bar{x}_i, \bar{u}_i)$ of the relaxation (4.5) by calling a

linear programming solver. This solution will necessarily satisfy (a), but it may violate (b) for some product i, for instance if $\bar{x}_i$ is not a permissible value of x_i, or $\bar{u}_i$ is not the correct value of $f_i(\bar{x}_i)$. The latter case is illustrated in Fig. 4.13, 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 optimization areas: (1) search by splitting intervals, from *continuous global optimization*, (2) domain reduction, from *constraint programming* and (3) polyhedral relaxation, from *integer programming*.

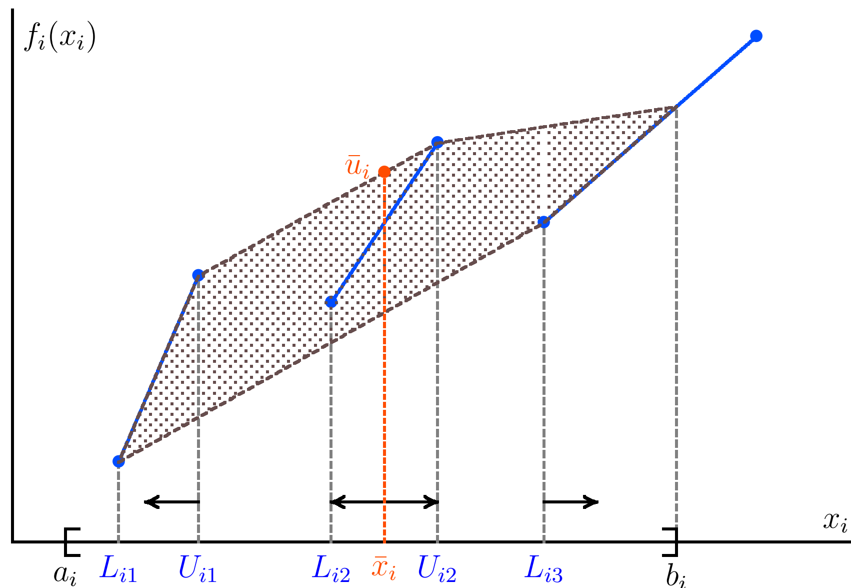


Figure 4.13: Production planning: convex hull relaxation (shaded area) of $f_i(x_i)$.

4.3.3 Experimental results

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. In order to break symmetry, the models also include constraints of the form $x_i \leq x_{i+1}$ for $i \in \{1, \ldots, n-1\}$. The number of search nodes and CPU time (in seconds) required to solve each of the ten

instances to optimality are shown in Table 4.1. As the number of products increases, one

Number of Products	IP (CPLEX 9.0) Searched nodes	IP (CPLEX 9.0) Time (s)	Integrated Searched nodes	Integrated 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**

Table 4.1: Production planning: search nodes and CPU time.

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. This can be explained by the fact that more powerful inference is applied before creating new nodes, and furthermore the search process uses more specific problem knoweldge (by exploiting the properties of the piecewise constraint). A MILP solver may find it very hard (and many times impossible) to uncover this information from a purely linear model. The integrated approach is also superior in all (but the three smallest) instances with respect to CPU time. The fact that CPLEX can solve the smaller IP instances faster than the integrated solver, despite the larger number of nodes explored, can be explained if we take into account the additional bookkeeping that goes on underneath the integrated solver at each node, like automatically reformulating (tightening the convex hull linearization of) the piecewise constraint. In general, the integrated solver will spend significantly more time at a node that a MILP solver does, but the hope is that this time is well spent and will reduce the number of nodes (and therefore the total solution time) considerably. This transpires quite clearly from Table 4.1, where the number of nodes required by the integrated solver does not grow nearly as fast as it does for the IP solver (in fact, a nice feature of this model seems to be that it results in a more robust behavior across problem instances, whereas the IP model's performance degrades significantly with problem size).

4.4 Example: Product configuration

In the second example we consider, the problem is to choose an optimal configuration of components for a product, such as a personal computer. For each component i, such as 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, $A_{ijk} < 0$ is used 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.

4.4.1 Modeling

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,

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

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 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.

A model based on metaconstraints

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.

A *variable index* is a very versatile modeling device. If an expression has the form u_y, where y is a variable, then y is a *variable index* or variable subscript. The modeling system automatically decodes variably indexed expressions with the help of the *element* metaconstraint, which is frequently used in constraint programming. In this case the variably indexed expression has the *indexed linear* form ua_y, where u is an integer variable and y a general discrete variable. An expression of this form is replaced with a new variable z and an additional constraint

$$\text{element}(y, (ua_1, \ldots, ua_n), z) \tag{4.8}$$

This constraint in effect forces $z = ua_y$. The solver can now apply a domain reduction (or filtering) algorithm to (4.8) and generate a relaxation for it.

4.4.2 Solution process

Filtering for (4.8) is straightforward. If z's domain is $[\underline{z}, \overline{z}]$, y's domain is D_y, and u's domain is $\{\underline{u}, \underline{u}+1, \ldots, \overline{u}\}$ at any point in the search, then the reduced domains $[\underline{z}', \overline{z}']$,

D'_y, and $\{\underline{u}', \ldots, \overline{u}'\}$ are given by

$$\underline{z}' = \min\left\{\underline{z}, \min_k \left\{a_k \underline{q}\right\}\right\}, \quad \overline{z}' = \max\left\{\overline{z}, \max_k \left\{a_k \overline{q}\right\}\right\},$$

$$D'_y = D_y \cap \left\{k \mid [\underline{z}', \overline{z}'] \cap [a_k \underline{q}, a_k \overline{q}] \neq \emptyset\right\}$$

$$\underline{u}' = \min\left\{\underline{u}, \min_k \left\{\lceil \underline{z}'/a_k \rceil\right\}\right\}, \quad \overline{u}' = \max\left\{\overline{u}, \max_k \left\{\lfloor \overline{z}'/a_k \rfloor\right\}\right\}$$

Inference via reformulation

Since (4.8) implies a disjunction $\bigvee_{k \in D_y}(z = a_k u)$, it can be given the standard convex hull relaxation for a disjunction, which in this case simplifies to

$$z = \sum_{k \in D_y} a_k u_k, \quad u = \sum_{k \in D_y} u_k$$

where $u_k \geq 0$ are new variables. Based on this idea, the relaxation of (4.7) becomes

$$\begin{aligned}
&\min \sum_j v_j \\
&v_j = \sum_i \sum_{k \in D_{t_i}} A_{ijk} q_{ik}, \quad \text{all } j \\
&q_i = \sum_{k \in D_{t_i}} q_{ik}, \quad \text{all } i \\
&L_j \leq v_j \leq U_j, \quad \text{all } j \\
&\underline{q}_i \leq q_i \leq \overline{q}_i, \quad \text{all } i \\
&q_{ik} \geq 0, \quad \text{all } i, k
\end{aligned} \tag{4.9}$$

Preventing future infeasibilities

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 (4.9). If $\hat{z} + r_{ik} \geq z^*$, where r_{ik} is the reduced cost of q_{ik} in the solution of (4.9), then k can be removed from the domain of t_i. In addition one can infer

$$\overline{q}_i \leq \min_{k \in D_{y_i}} \left\{\lfloor (z^* - \hat{z})/r_{ik} \rfloor\right\}, \quad \text{all } i$$

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

Search

The problem can be solved by branch and bound. In this case it is enough to branch 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, at which point the solution of the relaxation will necessarily be feasible.

4.4.3 Experimental results

For our computational experiments, we used ten of the problem instances proposed by Thorsteinsson and Ottosson[71], which have 26 components, up to 30 types per component, and 8 attributes. We also used the same branching strategy on q_i as they suggested: let $\bar{q}_i$ 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 each of the following constraints in turn (if possible): $q_i = \bar{q}_i$, $q_i \leq \bar{q}_i - 1$ and $q_i \geq \bar{q}_i + 1$. The number of search nodes and CPU time (in seconds) required to solve each of the ten

	IP Solver (CPLEX 9.0)		Integrated solver	
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

Table 4.2: Product configuration: search nodes and CPU time.

instances to optimality are shown in Table 4.2. Although the integrated model solves considerably slower than the pure MILP model, it again 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 production planning example. It is worth noting that THORSTEINSSON AND OTTOSSON[71] 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.

4.5 Example: Logic-based Benders decomposition

A third example we consider involves a decomposition approach for a machine scheduling problem. 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.

4.5.1 Modeling

An integer programming model can be written by discretizing time and letting $x_{ijt} = 1$ if job j starts processing at time t on machine i.

$$
\begin{aligned}
&\min \sum_{ijt} c_{ij} x_{ijt} \\
&\sum_{i,t} x_{ijt} = 1, \text{ all } i \\
&\sum_{j} \sum_{t' \in T_{ijt}} x_{ijt} \leq 1, \text{ all } i, t \\
&x_{ijt} = 0, \text{ all } i, j, t \text{ with } d_j - p_{ij} < t < r_j \text{ or } t > N - p_{ij} \\
&x_{ijt} \in \{0, 1\}
\end{aligned}
\tag{4.10}
$$

where N is the number of discrete times (beginning with time 0), and

$$T_{ijt} = \{t' \mid t - p_{ij} < t' \leq t\}$$

is the set of times at which a job j in progress on machine i at time t might start processing. There is also a continuous-time IP model suggested by TÜRKAY AND GROSSMANN[72], but computational testing (HOOKER[49]) indicates that it is much harder to solve than (4.10).

A model based on metaconstraints

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 be 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

$$\begin{aligned} &\min \sum_j c_{y_j j} \\ &r_j \leq t_j \leq d_j - p_{y_j j}, \text{ all } j \\ &\text{cumulative}((t_j | y_j = i), (p_{ij} | y_j = i), e, 1), \text{ all } i \end{aligned} \tag{4.11}$$

where e is a vector of ones.

4.5.2 Solution process

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 IP and a subproblem to be solved by CP. The assignment portion becomes the Benders master problem, which

allocates job j to machine i when $x_{ij} = 1$:

$$\begin{aligned} &\min \sum_{ij} c_{ij} x_{ij} \\ &\sum_i x_{ij} = 1, \text{ all } j \\ &\text{relaxation of subproblem} \\ &\text{Benders cuts} \\ &x_{ij} \in \{0, 1\} \end{aligned} \tag{4.12}$$

Inference

The solution $\bar{x}$ of the master problem determines the assignment of jobs to machines. Once these assignments are made, the problem (4.11) separates into a scheduling feasibility problem on each machine i:

$$\begin{aligned} &r_j \leq t_j \leq d_j - p_{\bar{y}_j j}, \text{ all } j \\ &\text{cumulative}((t_j | \bar{y}_j = i), (p_{ij} | \bar{y}_j = i), e, 1) \end{aligned} \tag{4.13}$$

where $\bar{y}_j = i$ when $\bar{x}_{ij} = 1$. If there is a feasible schedule for every machine, the problem is solved. If, however, the scheduling subproblem (4.13) is infeasible on some machine i, a Benders cut is generated to rule out the solution $\bar{x}$, 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 $\bar{x}$.

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

$$\sum_{j \in J_i} (1 - x_{ij}) \geq 1 \tag{4.14}$$

where $J_i = \{j \mid \bar{x}_{ij} = 1\}$. 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 examining the proof of infeasibility and noting which jobs actually play a role in the proof (see HOOKER[49]). 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.

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:

$$\sum_j p_{ij} x_{ij} \leq \max_j \{d_j\} - \min_j \{r_j\} \tag{4.15}$$

A Benders method (as well as any nogood-based method) fits easily into this unifying 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

$$\begin{array}{ll}
\min \sum_{ij} c_{ij} x_{ij} & (a) \\
\sum_i x_{ij} = 1, \text{ all } j & (b) \\
(x_{ij} = 1) \Leftrightarrow (y_j = i), \text{ all } i, j & (c) \\
r_j \leq t_j \leq d_j - p_{y_j j}, \text{ all } j & (d) \\
\text{cumulative}((t_j | y_j = i), (p_{ij} | \bar{y}_j = i), e, 1), \text{ all } i & (e)
\end{array} \tag{4.16}$$

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 master problem constraints (a) and (b) indicate that these constraints are themselves part of an IP relaxation of the problem. The relaxation parameter for (d) adds the inequalities (4.15) to the relaxation. The inference parameter for (e) specifies 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.

4.5.3 Experimental results

For our computational experiments, we used the instances proposed by JAIN AND GROSSMANN[53] and we created three additional instances with more than 20 jobs. These are the last instance in Table 4.3 and the last two instances in Table 4.4. Although state-of-the-art IP solvers have considerably improved since Jain and Grossmann's results were published, these instances are still intractable with the IP model (4.10). 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 IP model to the last instance of Table 4.3 had value 176, whereas the optimal solution has value 175. When processing times are shorter the problem tends to become easier, and we report

		IP Solver (CPLEX 9.0)		Integrated		
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**

Table 4.3: Parallel machine scheduling: long processing times.

results for this case in Table 4.4. Even here, the IP model is still much worse than the integrated Benders approach as the problem size grows. For instance, after 16.9 million search nodes the IP solver could not find a single feasible solution to the instance with 22 jobs and 5 machines. The best solution it found for the last instance, after 48 hours and 4.5 million search nodes, had value 181, and the optimal value is 179.

		IP Solver (CPLEX 9.0)		Integrated		
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**

Table 4.4: Parallel machine scheduling: short processing times.

Chapter 5

Applications

5.1 The robust spanning tree problem

Given an undirected graph with interval edge costs, a robust spanning tree is one whose cost is as close as possible to that of a minimum spanning tree under any possible assignment of costs. This section defines the problem formally and introduces the main concepts and notations necessary to understand the proposed solution.

Let $G = (V, E)$ be an undirected graph with $|V| = n$ nodes and $|E| = m$ edges and an interval $[\underline{c_e}, \overline{c_e}]$ for the cost of each edge $e \in E$. A **scenario** s is a particular assignment of a cost $c_e \in [\underline{c_e}, \overline{c_e}]$ to each edge $e \in E$. We use c_e^s to denote the cost of edge e under a given scenario s. Recall that a *spanning tree* for $G = (V, E)$ is a set of edges $T \subseteq E$ such that the subgraph $G' = (V, T)$ is acyclic and $\forall\ i \in V,\ \exists\ j \in V :\ (i, j) \in T$. The cost of a spanning tree T for a scenario s, denoted by c_T^s, is the sum of the costs of all edges under scenario s:

$$c_T^s = \sum_{e \in T} c_e^s.$$

A minimum spanning tree for scenario s, denoted by MST^s, is a spanning tree with minimal cost for scenario s and its cost is denoted by c_{MST^s}. Following Yaman et al[76], we now define the worst case scenario for a spanning tree T and the robust deviation of T, two fundamental concepts for this section.

Definition 5.1.1 (Relative worst case scenario)**.** Given a spanning tree T, a scenario $w(T)$ which maximizes the difference between the cost of T and the cost of a minimum spanning tree under $w(T)$ is called a *relative worst case scenario* for T. More precisely, a relative worst case scenario $w(T)$ satisfies

$$w(T) \in \text{arg-}\max_{s \in \mathcal{S}} (c_T^s - c_{MST^s}) \tag{5.1}$$

where $\mathcal{S}$ is the set of all possible scenarios.

Note that $\text{arg-}\max_{s \in \mathcal{S}} \; f(s)$ denotes the set

$$\{e \in S \;\mid\; f(e) = \max_{s \in S} f(s)\}.$$

and arg-min is defined similarly.

Definition 5.1.2 (Robust Deviation)**.** The robust deviation of a spanning tree T, denoted by Δ_T, is the distance between the cost of T and the cost of a minimum spanning tree under the relative worst case scenario of T:

$$\Delta_T = c_T^{w(T)} - c_{MST^{w(T)}}. \tag{5.2}$$

The goal is to compute a robust spanning tree, i.e., a spanning tree whose robust deviation is minimal.

Definition 5.1.3 (Robust Spanning Tree)**.** A (relative) robust spanning tree is a spanning tree T^* whose robust deviation is minimal, i.e.,

$$T^* \in \text{arg-}\min_{T \in \Gamma_G} \max_{s \in \mathcal{S}} (c_T^s - c_{MST^s})$$

where Γ_G is the set of all spanning trees of G and $\mathcal{S}$ is the set of all possible scenarios. According to the definition of $w(T)$ (Eq. 5.1), this is equivalent to:

$$T^* \in \text{arg-}\min_{T \in \Gamma_G} (c_T^{w(T)} - c_{MST^{w(T)}}).$$

Figure 5.1 depicts these concepts graphically. The horizontal axis depicts various scenarios. For each scenario s, it shows the cost c_T^s of T under s and the cost c_{MST^s} of an MST under s. In the figure, scenario s_5 is a worst-case scenario, since the distance $c_T^{s_5} - c_{MST^{s_5}}$

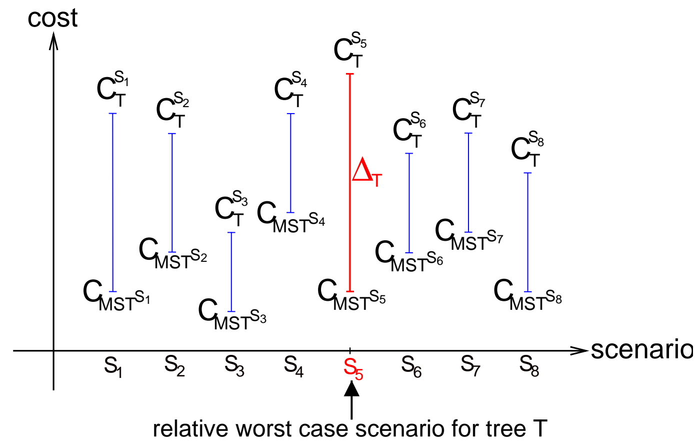


Figure 5.1: Robust spanning tree: robust deviation

is maximal and this distance is then the robust deviation of T. Such a figure can be drawn for each spanning tree and we are interested in finding the spanning tree with the smallest robust deviation. KOUVELIS AND YU[59] conjectured that the robust spanning tree problem with interval edges is NP-complete, and ARON AND VAN HENTENRYCK[5] proved that this conjecture was true.

Prior Work

YAMAN ET AL[76] proposed an elegant MIP formulation for the RSTPIE problem. The formulation combines the single commodity model of the minimum spanning tree with the dual of the multicommodity model for the same problem. In addition, they introduced the concept of weak and strong edges and used them for preprocessing. They showed that preprocessing significantly enhances the performance of their MIP implementation. We now summarize their relevant results.

We first introduce the concept of weak edges, i.e., the only edges that need to be considered during the search.

Definition 5.1.4 (Weak Tree)**.** A tree $T \subseteq E$ is weak if there exists at least one scenario under which T is a minimum spanning tree of G.

Definition 5.1.5 (Weak Edge)**.** An edge $e \in E$ is *weak* if it lies on at least one weak tree.

Strong edges are edges that are necessarily part of a robust spanning tree.

Definition 5.1.6 (Strong Edge)**.** An edge $e \in E$ is *strong* if it lies on a minimum spanning tree of G for all possible scenarios.

The following propositions characterize weak and strong edges in terms of the minimum spanning trees of two scenarios.

Proposition 5.1.7. *An edge $e \in E$ is weak if and only if there exists a minimum spanning tree using edge e when its cost is at the lowest bound and the costs of the remaining edges are at their highest bounds.*

Proposition 5.1.8. *An edge $e \in E$ is strong if and only if there exists a minimum spanning tree using edge e when its cost is at the highest bound and the costs of the remaining edges are at their lowest bounds.*

As a consequence, YAMAN ET AL[76] showed that it is possible to use a slightly modified version of Kruskal's algorithm to find all the weak and strong edges of a given graph in $O(m^2 \log m)$ time. In Section 5.1.1, we give an improved algorithm for finding weak edges, which runs in $O(m \log m)$. The following two propositions capture the intuition we gave earlier on weak and strong edges.

Proposition 5.1.9. *A relative robust spanning tree T is a weak tree. Thus an edge can be part of a relative robust spanning tree only if it is a weak edge.*

Proposition 5.1.10. *There exists a relative robust tree T such that every strong edge in the graph lies on T.*

The next result is also fundamental: it makes it possible to characterize precisely the worst case scenario of a spanning tree T.

Proposition 5.1.11. *The scenario in which the costs of all edges in a spanning tree T are at upper bounds and the costs of all other edges are at lower bounds is a relative worst case scenario for T. In other words, $w(T)$ is specified as*

$$c_e^{w(T)} = \begin{cases} \overline{c_e}, & e \in T \\ \underline{c_e}, & e \in E \setminus T \end{cases} \tag{5.3}$$

In other words, once we select a tree T, we can easily find the worst-case scenario for T by assigning to the edges in T their upper bounds and to the edges not in T their lower bounds. We use Proposition 5.1.11 in our new algorithm. In the rest of this section, we also use the notation $w(S)$ to denote the scenario where $c_e = \overline{c_e}$ if $e \in S$ and $c_e = \underline{c_e}$ otherwise even when S is not a tree.

Solution process

```
Algorithm Search (⟨S, R⟩)

1   if (|S| < n − 1) then
2       ⟨S, R⟩ ← PruneInfeasible (⟨S, R⟩)
3       ⟨S, R⟩ ← PruneSuboptimal (⟨S, R⟩)
4       if (LB(⟨S, R⟩) ≤ f*) then
5           e ← SelectEdge (E \ (S ∪ R))
6           Search (⟨S, R ∪ { e }⟩)
7           Search (⟨S ∪ { e } , R⟩)
8       else
9           if (Δ_S < f*) then
10              T* ← S
11              f* ← Δ_S
12          end if
13      end if
14  end if
```

Figure 5.2: Robust spanning tree: the search algorithm

Figure 5.2 gives a high-level description of the search algorithm. A node in the search tree is called a configuration to avoid confusion with the nodes of the graph. A configuration is a pair $\langle S, R\rangle$, where S represents the set of selected edges and R represents the set of rejected edges. The algorithm receives a configuration as input. If the configuration is not a spanning tree, the algorithm prunes infeasible and suboptimal edges. Both steps remove edges from $E \setminus (S \cup R)$ and adds them to R. If the lower bound of the resulting configuration is smaller than the best found solution, the algorithm selects an edge and explores two subproblems recursively. The subproblems respectively reject and select the

Notation	Definition
c^s_T	the cost of tree T under scenario s $c^s_T = \sum_{e \in T} c^s_e$
$w(T)$	worst case scenario for edge set T: $c_e^{w(T)} = \begin{cases} \overline{c_e}, & e \in T \\ \underline{c_e}, & e \in E \setminus T \end{cases}$
Γ_G	the set of all spanning trees of graph G
$\mathcal{T}(S,R)$	the set of spanning trees derived from $\langle S, R \rangle$ $\mathcal{T}(S,R) = \{ T \in \Gamma_G \| S \subseteq T \subseteq E \setminus R \}$
$\mathcal{M}^s(S,R)$	the set of spanning trees from $\mathcal{T}(S,R)$ with mimimal cost under scenario s $\mathcal{M}^s(S,R) = \text{arg-}\min_{T \in \mathcal{T}(S,R)} c^s_T$
$MST^s(S,R)$	an arbitrary tree from $\mathcal{M}^s(S,R)$
$\mathcal{M}^s$	$\mathcal{M}^s(\emptyset, \emptyset)$
MST^s	an arbitrary tree from $\mathcal{M}^s(\emptyset, \emptyset)$

Table 5.1: Robust spanning tree: notations

edge. The best found solution and upper bound are updated each time a spanning tree with a smaller robust deviation is obtained.

5.1.1 Inference

Infeasibility pruning is relatively simple in our algorithm. It ensures that S can be extended into a spanning tree and removes edges that would create cycles. We do not discuss infeasibility further and instead focus on the lower bound and suboptimality pruning. Before doing so, we introduce some additional notations. We use $\mathcal{T}(S,R)$ to denote the set of all spanning trees that can be derived from configuration $\langle S, R \rangle$, i.e.,

$$\mathcal{T}(S,R) = \{T \in \Gamma_G \;\mid\; S \subseteq T \subseteq E \setminus R\}.$$

Given a scenario s, $\mathcal{M}^s(S,R)$ denotes the set of all minimum spanning trees which are derived from configuration $\langle S, R \rangle$ under scenario s and $MST^s(S,R)$ is a representative of $\mathcal{M}^s(S,R)$. For simplicity, we use $\mathcal{M}^s$ and MST^s when $S = \emptyset$ and $R = \emptyset$. All relevant notations are summarized in Table 5.1 for convenience.

The Lower Bound

We now present a lower bound to the robust deviation of any spanning tree derived from

$\langle S, R\rangle$. In other words, we need to find a lower bound on the value of Δ_T (as defined by Eq. 5.2), for any tree $T \in \mathcal{T}(S, R)$. Recall that, for any such tree T, the robust deviation is given by $\Delta_T = c_T^{w(T)} - c_{MST^{w(T)}}$. We can approximate Δ_T by finding a lower bound to $c_T^{w(T)}$ and an upper bound to $c_{MST^{w(T)}}$. Since $S \subseteq T \subseteq S \cup L$ where $L = E \setminus (S \cup R)$, both bounds can be obtained by considering the scenario $w(S \cup L)$. As a consequence, we define the lower bound $LB(\langle S, R\rangle)$ of a configuration $\langle S, R\rangle$ as

$$LB(\langle S, R\rangle) = c_{MST^{w(S\cup L)}(S,R)} - c_{MST^{w(S\cup L)}}.$$

The following proposition proves that $LB(\langle S, R\rangle)$ is indeed a lower bound.

Proposition 5.1.12. *Let $\langle S, R\rangle$ be an arbitrary configuration and let $L = E \setminus (S \cup R)$. Then, for all $T \in \mathcal{T}(S, R)$, we have*

$$\Delta_T \geq c_{MST^{w(S\cup L)}(S,R)} - c_{MST^{w(S\cup L)}}.$$

Proof: Since $T \in \mathcal{T}(S, R)$, it follows that $S \subseteq T \subseteq E - R = S \cup L$. Therefore,

$$c_{MST^{w(T)}} \leq c_{MST^{w(S\cup L)}}$$

On the other hand, by the definition of a minimum spanning tree and since $T \subseteq S \cup L$, we have that

$$c_{MST^{w(S\cup L)}(S,R)} \leq c_T^{w(S\cup L)} = c_T^{w(T)}.$$

The result follows since

$$c_{MST^{w(S\cup L)}(S,R)} - c_{MST^{w(S\cup L)}} \leq c_T^{w(T)} - c_{MST^{w(T)}} = \Delta_T.$$

Observe that this lower bound only requires the computation of two minimum spanning trees and hence it can be computed in $O(m \log m)$ time. Interestingly, $c_{MST^{w(S\cup L)}}$ can use any edge in the graph and is thus independent of the edges selected in S and R. Of course, the scenario $w(S \cup L)$ is not! It is easy to see that this lower bound is monotone in both arguments

$$LB(\langle S, R\rangle) \leq LB(\langle S \cup \{e\}, R\rangle)$$
$$LB(\langle S, R\rangle) \leq LB(\langle S, R \cup \{e\}\rangle).$$

Suboptimality Pruning

A fundamental component of our algorithm is suboptimality pruning, i.e., the ability to remove all non-weak edges at every configuration of the search tree. The results in YAMAN ET AL[76] allow us to detect all weak edges in time $O(m^2 \log m)$ by solving m MSTs. This cost is prohibitive in practice. We now show how to detect all weak edges by solving a single MST and performing a postprocessing step for each edge. The overall complexity is $O(n^2 + m \log m)$, which is $O(m \log m)$ on dense graphs.

The key idea is to characterize all weak edges in terms of a unique scenario. The characterization is based on the following results, which specify when a tree remains an MST under cost changes and the cost of an MST which must contain a specified edge.

Proposition 5.1.13. *Let s be a scenario and $T \in \mathcal{M}^s$. Let $e = (u, v) \notin T$ and $f = (x, y)$ be the edge of maximal cost on the path from u to v. Consider $\hat{s}$ the scenario s where c_e^s is replaced by $c_e^{\hat{s}}$, all other costs remaining the same. Then $T \in \mathcal{M}^{\hat{s}}$ if $c_e^{\hat{s}} \geq c_f^{\hat{s}}$.*

Proof: By contradiction. Assume that there exists a tree T' containing e such that $c_{T'}^{\hat{s}} < c_T^{\hat{s}}$. Removing e from T' produces two connected components C_1 and C_2. If $x \in C_1$ and $y \in C_2$, then we can construct a tree

$$T'' = T' \setminus \{e\} \; \cup \{f\}$$

and we have

$$c_{T''}^{\hat{s}} = c_{T'}^{\hat{s}} - (c_e^{\hat{s}} - c_f^{\hat{s}}) < c_{T'}^{\hat{s}} < c_T^{\hat{s}}.$$

Since $e \in T''$ and $e \in T$, we have

$$c_{T''}^{s} = c_{T''}^{\hat{s}} < c_T^{\hat{s}} = c_T^{s}$$

which contradicts the fact that $T \in \mathcal{M}^s$. If $x, y \in C_1$ (resp. C_2), since there exists a cycle in the graph containing e and f, there exists at least one edge g on the path from u to v in T such that $g \in T'$ (otherwise T' would not be a tree). By hypothesis, $c_g^s \leq c_f^s$ and hence $c_e^{\hat{s}} \geq c_g^{\hat{s}}$. We can thus apply the same construction as in the case $x \in C_1$ and $y \in C_2$ with f replaced by g.

Proposition 5.1.14. *Let s be a scenario and T be an MST^s. Let $e = (u, v) \notin T$ and $f = (x, y)$ be the edge of maximal cost on the path from u to v. Then, $T \setminus \{f\} \cup \{e\} \in \mathcal{M}^s(\{e\}, \emptyset)$.*

The proof of this result is similar to the proof of Proposition 5.1.13. We are now ready to present a new characterization of weak edges. The characterization only uses the scenario $\bar{s}$ where all costs are at their upper bounds.

Proposition 5.1.15. *Let $\bar{s}$ be the scenario where all costs are at their upper bounds and $T \in \mathcal{M}^{\bar{s}}$. An edge $e = (u, v)$ is weak if $e \in T$ or if an edge f of maximal cost on the path from u to v in T satisfies $\underline{c_e} \leq \overline{c_f}$.*

Proof: Let $\hat{s}$ the scenario $\bar{s}$ where $c_e^{\bar{s}}$ is replaced by $c_e^{\hat{s}}$. If $e \in T$, then $T \in \mathcal{M}^{\hat{s}}$ as well and hence e is weak by Proposition 5.1.7. If $e \notin T$ and $\underline{c_e} > \overline{c_f}$, then $T \in \mathcal{M}^{\hat{s}}$ by Proposition 5.1.13. By Proposition 5.1.14, $T \setminus \{f\} \cup \{e\} \in \mathcal{M}^{\bar{s}}(\{e\}, \emptyset)$ and its cost is greater than c_T since $\underline{c_e} > \overline{c_f}$. Hence e is not weak. If $e \notin T$ and $\underline{c_e} = \overline{c_f}$, then $T \setminus \{f\} \cup \{e\}$ is an $MST^{\bar{s}}$ and hence e is weak. The same holds for the case where $e \in T$ and $\underline{c_e} < \overline{c_f}$.

```
Algorithm MaxCost (u, v)

1   if (u = v) then
2       return 0
3   else
4       if (level(u) < level(v)) then
5           return max{ cost(v, p(v)) , MaxCost (u, p(v)) }
6       else
7           if (level(u) > level(v)) then
8               return max{ cost(u, p(u)) , MaxCost (p(u), v) }
9           else
10              maxedge ← max{ cost(u, p(u)) , cost(v, p(v)) }
11              return max{ maxedge , MaxCost (p(u), p(v)) }
12          end if
13      end if
14  end if
```

Figure 5.3: Robust spanning tree: finding the largest cost of an edge

We now describe how to use Proposition 5.1.15 to obtain an $O(m \log m)$ algorithm on dense graphs. The key idea is to compute $MST^{\bar{s}}$, i.e., the MST when all costs are at

their upper bounds. All edges in this MST are weak. In addition, for each $e = (u, v)$ not in the MST, we compute the largest cost of any edge on the path from u to v. This can be done easily by upgrading Prim's algorithm slightly to associate a level and a parent to each vertex. When an edge $e = (u, v)$ with $u \in T$ and $v \notin T$ is added to T in Prim's algorithm, we set

```
level(v) = level(u) + 1;
parent(v) = u;
```

These modifications do not affect the complexity of the algorithm. It now suffices to apply the algorithm depicted in Figure 5.3 to compute the cost of the maximal edge and to apply Proposition 5.1.15. The algorithm in Figure 5.3 simply follows the paths from u and v to their common ancestor, computing the cost of the maximal edge on the way. Since algorithm `MaxCost` takes $O(n)$ in the worst case, the overall complexity becomes $O(m \log m + mn)$. However, it is easy to reduce this complexity to $O(m \log m + n^2)$ by amortizing the computation of the maximal costs. It suffices to cache the results of `MaxCost` in an $n \times n$ matrix M. For each edge $e = (u, v) \in E \setminus T$ we first examine entry $M[u, v]$ and call `MaxCost`(u,v) only if this entry is uninitialized. Since there are at most n^2 entries and each call to `MaxCost`(u,v) costs $O(1)$ per entry it fills, the overall complexity complexity becomes $O(m \log m + n^2)$. We proved the following theorem.

Theorem 5.1.16. *All weak edges of a graph can be computed in $O(m \log m + n^2)$ time.*

5.1.2 Search

It is well-known that a good branching heuristic may improve performance significantly. We now show a branching heuristic adapting the first-fail principle (see Haralick and Elliott[41]) to robust optimization. The key idea is to explore the most uncertain edges first, i.e., to branch on an edge e with the maximal difference $\overline{c_e} - \underline{c_e}$. Indeed, rejecting e (i.e., adding e to R) allows $MST^{w(S \cup L)}$ to select e at a low cost, possibly giving a large deviation. Hence this branch is likely to fail early. However, this is only important if $MST^{w(S \cup L)}$ is likely to select e which may not necessarily the case if $\underline{c_e}$ is large compared to

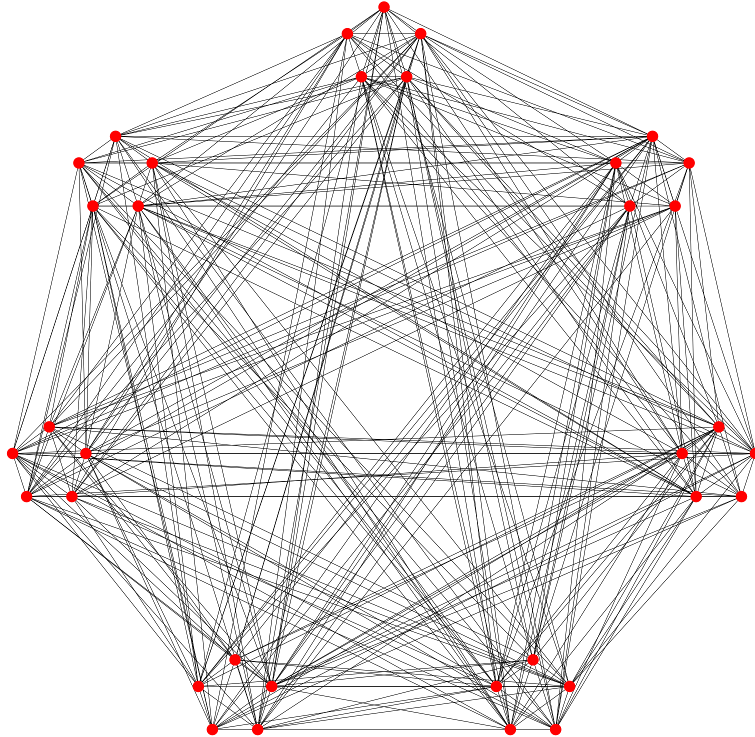


Figure 5.4: Robust spanning tree: a class 7 network

other "similar" edges. Hence, it seems appropriate to select first an edge $e \in MST^{w(S \cup L)}$ whose difference $\overline{c_e} - \underline{c_e}$ is maximal. This justifies the following branching strategy.

Definition 5.1.17 (Branching Strategy)**.** Let $\langle S, R \rangle$ be a configuration, $L^* = L \cap MST^{w(S \cup L)}$, and $L^- = L \setminus MST^{w(S \cup L)}$. The branching strategy selects an edge s defined as follows:

$$s = \begin{cases} \text{arg-}\max_{e \in L^*} \; \overline{c_e} - \underline{c_e} & \text{if } L^* \neq \emptyset \\ \text{arg-}\max_{e \in L^-} \; \overline{c_e} - \underline{c_e} & \text{otherwise.} \end{cases}$$

5.1.3 Experimental results

We now report experimental results comparing the constraint satisfaction and the MIP approaches. The comparison uses the instances in Yaman et al[76], as well as some new instances which capture additional structure arising in practical applications.

The experimental setting of Yaman et al[76] uses complete graphs with n vertices and six classes of problems. Three of the six classes use tight intervals for edge costs,

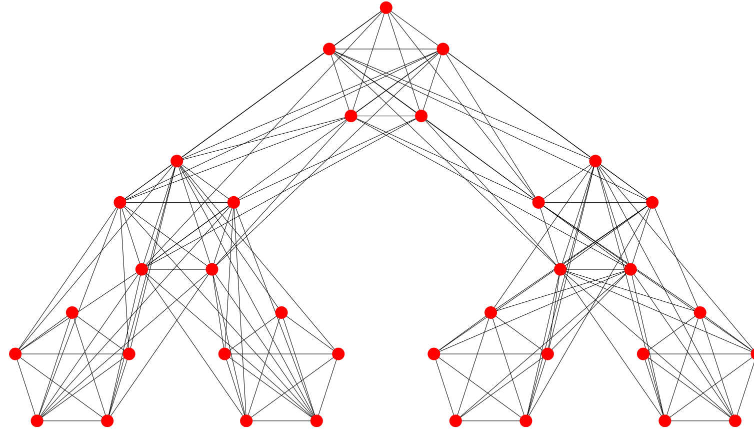


Figure 5.5: Robust spanning tree: a class 8 network

while the other three allowed larger differences between the lower and upper bounds. The edge intervals were chosen as follows. The values of $\underline{c_e}$ and $\overline{c_e}$ are uniformly distributed in the intervals:

class 1: $\underline{c_e} \in [0, 10]$, $\overline{c_e} \in (\underline{c_e}, 10]$

class 2: $\underline{c_e} \in [0, 15]$, $\overline{c_e} \in (\underline{c_e}, 15]$

class 3: $\underline{c_e} \in [0, 20]$, $\overline{c_e} \in (\underline{c_e}, 20]$

class 4: $\underline{c_e} \in [0, 10]$, $\overline{c_e} \in (\underline{c_e}, 20]$

class 5: $\underline{c_e} \in [0, 15]$, $\overline{c_e} \in (\underline{c_e}, 30]$

class 6: $\underline{c_e} \in [0, 20]$, $\overline{c_e} \in (\underline{c_e}, 40]$

Note that the size of the search space to explore is $O(2^{300})$ for a complete graph of 25 nodes. Of course, our constraint satisfaction algorithm will only explore a small fraction of that space. In addition to these six classes, we also generate instances (Classes 7 and 8) whose cost structure is not the same for all the edges. Class 7 contains instances which represent a two-level network. The lower level consists of clusters of 5 nodes whose edges are generated according to Class 1 above. The upper level links the clusters and these edges have higher costs, i.e., Class 1 costs shifted by a constant which is larger than the Class 1 edges. This captures the fact that, in networks, there are often various types of edges with different costs. See Figure 5.4 for an instance of Class 7 with 35 nodes. Class 8 contains instances which are similar to Class 7, except that the upper-level layer is organized as a binary tree. See Figure 5.5 for an instance of Class 8 with 35 nodes. In

Edges(Nodes)	Algo.	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8
45(10)	CSR	0.12	0.07	0.08	0.08	0.06	0.07	0.06	0.06
	CSF	0.14	0.10	0.12	0.12	0.09	0.11	0.07	0.07
	MIP	6.59	3.95	4.10	4.38	4.68	3.83	2.29	2.02
105(15)	CSR	1.82	1.96	1.09	0.36	0.86	0.33	1.76	1.15
	CSF	2.53	3.06	2.15	0.86	2.01	0.77	1.83	1.35
	MIP	245.19	184.14	136.88	109.30	91.33	87.62	730.20	88.66
190(20)	CSR	39.72	33.80	8.91	3.43	2.09	3.12	7.37	9.51
	CSF	74.86	61.08	12.66	7.56	4.69	7.05	5.28	6.77
	MIP	8620.66	5517.75	14385.55	3344.95	12862.29	22855.82	18547.27	1399.48
300(25)	CSR	121.85	181.57	68.42	20.41	12.23	13.26	91.89	61.71
	CSF	244.18	272.63	145.11	50.37	30.55	32.41	97.94	36.21
435(30)	CSR	926.65	415.07	942.38	133.90	63.85	177.47	1719.61	721.28
	CSF	2100.43	909.15	1811.88	359.72	167.58	418.41	804.78	284.39
595(35)	CSR	4639.55	5095.71	2304.36	383.34	188.36	419.56	32511.77	6356.78
	CSF	10771.37	11906.01	4183.22	1100.88	548.01	1115.83	14585.73	2149.58
780(40)	CSR	27206.38	16388.12	15059.39	1103.50	1122.57	1071.62	57309.23	33390.67
	CSF	29421.70	34666.84	22084.00	3456.84	3241.04	3031.55	28432.91	11339.59

Table 5.2: Robust spanning tree: average CPU time

general, classes 7 and 8 are significantly harder than classes 1 to 6, since preprocessing is less effective than in Classes 1 to 6 because of the additional structure. Observe also that these instances are sparser for the same number of nodes.

Table 5.2 compares the efficiency of the two approaches. It reports the computation times in CPU seconds of the MIP implementations, the constraint satisfaction algorithm with suboptimality pruning at the root node only (CSR), and the constraint satisfaction algorithm with suboptimality pruning at every node (CSF). The times are given on a Sun Sparc 440Mhz processor and the average is computed over 10 instances for each class and each size. We used CPLEX 6.51 for solving the MIP, after preprocessing of the weak and strong edges. Note that the constraint programming approach produces dramatic speedups over the pure integer programming approach. On Class 1, CSR runs about 134 times faster than the MIP on graphs with 15 nodes and about 217 times faster on graphs with 20 nodes. On Classes 7 and 8, the speed-ups are even more impressive. On graphs with 20 nodes for classes 7 and 8, CSR runs about 3500 and 200 times faster than the MIP. (Recall that these graphs are not complete). The MIP times are not given for graphs with more than 20 nodes, since they cannot be obtained in reasonable time. The results indicate that the constraint satisfaction approach is also better at exploiting the network structure, which is likely to be fundamental in practice. Observe also that the constraint

satisfaction approach is able to tackle much large instances in reasonable times.

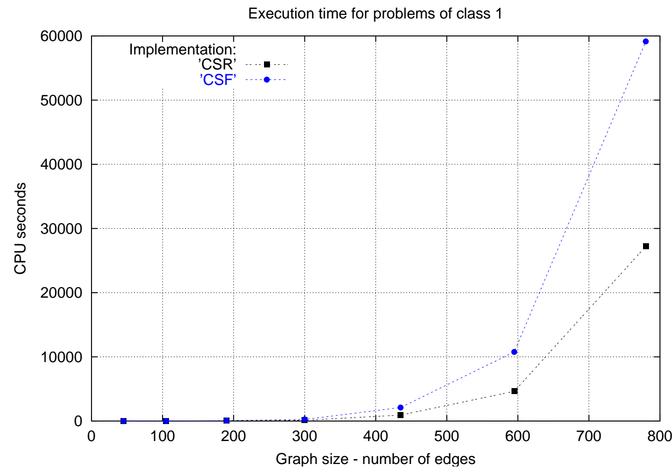


Figure 5.6: Robust spanning tree: results using CP on class 1.

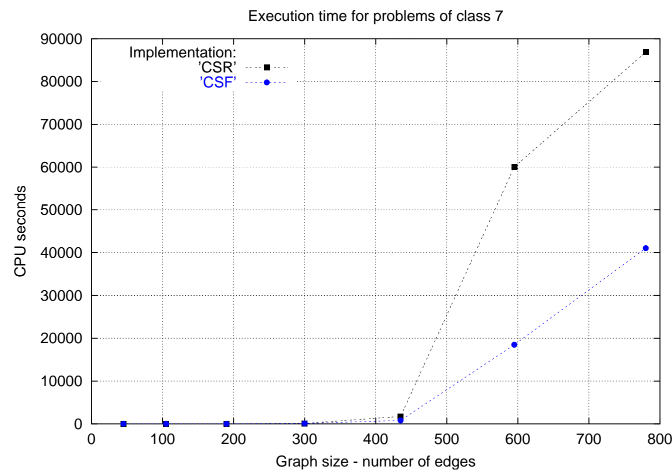


Figure 5.7: Robust spanning tree: results using CP on class 7.

Figures 5.6, 5.7, and 5.8 plot the execution times of the two constraint satisfaction algorithms. Observe also that applying suboptimality pruning at every node is not cost-effective on Classes 1 to 6. This is due to the fact that graphs are complete and costs are uniformly distributed in these instances. Classes 7 and 8, which add a simple additional cost structure, clearly indicate that suboptimality pruning becomes increasingly important when the network is more heterogeneous. The benefits of suboptimality pruning clearly appears on large graphs for class 8, where CSF is about three times as fast in the average.

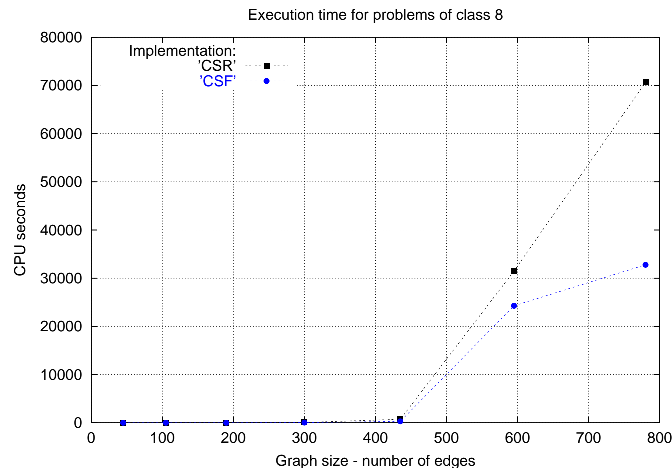


Figure 5.8: Robust spanning tree: results using CP on class 8.

In fact, CSF is significantly faster (e.g., 10 times faster) than CSR on some instances, while the two algorithms are similar on others. *It is important to mention that systematic suboptimality pruning is useless without Theorem 5.1.16. Indeed, the pruning benefit is often offset by the high pruning cost otherwise.*

Overall, it is clear that the constraint satisfaction approach is much more effective on these problems than the MIP approach. It produces extremely dramatic speed-ups and substantially enlarge the class of instances that are amenable to effective solutions. The constraint satisfaction approach is able to solve large-scale problems (over 1,000 edges). Since networks are often organized in hierarchies, it should scale up nicely to larger real-life instances. There is still considerable room for improvement in the implementation, since incremental MST algorithms (see RAUCH ET AL[68]) and enhanced feasibility pruning may decrease runtime significantly.

Bibliography

[1] Achterberg, T. SCIP: A framework to integrate constraint and mixed integer programming. Tech. Rep. 04-19, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2004.

[2] Ajili, F., and Wallace, M. Hybrid problem solving in ECLiPSe. In *Constraint and Integer Programming: Toward a Unified Methodology*, M. Milano, Ed. Kluwer, 2003, pp. 169–201.

[3] Applegate, D., Bixby, R., Chvatal, V., and Cook, B. Finding cuts in the tsp (a preliminary report). Tech. Rep. 95-05, DIMACS, 1995.

[4] Aron, I., and Hentenryck, P. V. A constraint satisfaction approach to the robust spanning tree problem with interval data. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence UAI* (Aug. 1–4 2002).

[5] Aron, I., and Hentenryck, P. V. On the complexity of the robust spanning tree problem with interval data. *Operations Research Letters (ORL) 32* (Jan. 2004), 36–40.

[6] Aron, I. D., Hooker, J. N., and Yunes, T. H. SIMPL: A system for integrating optimization techniques. *Lecture Notes in Computer Science 3011* (2004), 21–36.

[7] Balaş, E. A modified lift-and-project procedure. *Math Programming 79* (1997), 19–31.

[8] Balaş, E., and Perregaard, M. A precise correspondence between lift-and-project cuts, simple disjunctive cuts, and mixed integer gomory cuts for 0-1 programming. *Mathematical Programming* (2002).

[9] Balas, E. Intersection cuts: a new type of cutting planes for integer programming. *Operations Research 19* (1971), 19–39.

[10] Balas, E. Disjunctive programs: Cutting planes from logical conditions. In *Nonlinear Programming*, O. L. M. et al., Ed., vol. 2. Academic Press, 1975, pp. 279–312.

[11] Balas, E. Disjunctive programming. *Annals of Discrete Mathematics 5* (1979), 3–51.

[12] Balas, E. Disjunctive programming: properties of the convex hull of feasible points. *Discrete Applied Mathematics 89* (1998), 1–44. Originally MSRR 348, Carnegie Mellon University, 1974.

[13] Balas, E. Generating deepest mixed integer cuts by iterative disjunctive modularization. Working paper, Carnegie Mellon University, GSIA, May 2002.

[14] Balas, E., Ceria, S., and Cornuejols, G. A lift-and-project cutting plane algorithm for mixed 0-1 programming. *Mathematical Programming 58* (1993), 295–324.

[15] Balas, E., Ceria, S., and Cornuejols, G. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science 42* (1996), 1229–1246.

[16] Balas, E., and Jeroslow, R. G. Strengthening cuts for mixed integer programs. *European Journal of Operations Research 4* (1980), 224–234.

[17] Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., and Vance, P. H. Branch-and-price: column generation for solving huge integer programs. *Operations Research 46* (1998), 316–329.

[18] Barth, P., and Bockmayr, A. PLAM: ProLog and Algebraic Modelling. In *Proceedings of the Fifth International Conference on the Practical Application of Prolog* (London, UK, 1997), The Practical Application Company, pp. 73–82.

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

[20] Beldiceanu, N. Global constraints catalog, 2006. http://www.emn.fr/x-info/sdemasse/gccat/.

[21] Berge, C. *Graphs and Hypergraphs*. North Holland, 1973.

[22] Bisschop, J., and Entriken, R. Aimms: The modeling system. Tech. rep., Paragon Decision Technology, 1993.

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

[24] Brooke, A., Kendrick, D., and Meeraus, A. *GAMS: A User's Guide*. The Scientific Press, 1992.

[25] Caprara, A., and Fischetti, M. Branch and cut algorithms. In *Annotated bibliographies in Combinatorial Optimization*, M. Dell'Amico and S. M. F. Maffioli, Eds. John Wiley & Sons, 1997.

[26] Ceria, S., and Pataki, G. Solving integer and disjunctive programs by lift and project. *Lecture Notes in Computer Science 1412* (1998), 271–283.

[27] Ceria, S., and Soares, J. Disjunctive cut generation for mixed 0-1 programs: duality and lifting. *Graduate School of Business, Columbia University* (1997).

[28] Chandru, V., and Hooker, J. N. *Optimization Methods for Logical Inference*. Wiley, 1999.

[29] Ciriani, T. A., Colombani, Y., and Heipcke, S. Embedding optimisation algorithms with mosel. *Journal 4OR: A Quarterly Journal of Operations Research 1*, 2 (June 2003), 155–167.

[30] Danzig, G. Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production an Allocation*, T. Koopmans, Ed. Wiley, 1951, ch. 21, pp. 339–347.

[31] Dash Optimization. *Xpress-MP User Manual*. http://www.dashoptimization.com, 2003.

[32] Dash Optimization. *Xpress-Kalis User Manual*. http://www.dashoptimization.com, 2006.

[33] Focacci, F., Lodi, A., and Milano, M. Solving tsp through the integration of or and cp techniques. *Workshop on Large Scale Combinatorial Optimization and Constraints* (1998).

[34] Focacci, F., Lodi, A., and Milano, M. Cost-based domain filtering. In *CP '99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming* (London, UK, 1999), Springer-Verlag, pp. 189–203.

[35] Focacci, F., Lodi, A., and Milano, M. Integration of CP and OR methods for matching problems. *CP-AI-OR'99 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems.* (1999).

[36] Fourer, R., Gay, D., and Kernighan, B. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, 1993.

[37] Glover, F. Convexity cuts and cut search. *Operations Research 21* (1973), 123–134.

[38] Gomory, R. E. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society 64* (1958), 275–278.

[39] HAKEN, A. The intractability of resolution. *Theoretical Computer Science 39* (1985), 297–308.

[40] HAMMER, P. L., AND RUDEANU, S. Boolean methods in operations research and related areas, 1968.

[41] HARALICK, R., AND ELLIOT, G. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14* (1980), 263–313.

[42] HARALICK, R. M., AND ELLIOTT, G. L. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14* (1980), 263–313.

[43] HENTENRYCK, P. V., LUSTIG, I., AND MICHEL, L. *Numerica: A modeling language for global optimization*. MIT Press, 1997. ISBN 0-262-72027-2.

[44] HENTENRYCK, P. V., LUSTIG, I., AND MICHEL, L. *The OPL Optimization Programming Language*. MIT Press, 1999. ISBN 0-262-72030-2.

[45] HENTENRYCK, P. V., AND MICHEL, L. Helios: A modeling language for nonlinear constraint solving and global optimization using interval analysis. Tech. Rep. CS-95-33, Department of Computer Science, Brown University, 1995.

[46] HENTENRYCK, P. V., MICHEL, L., AND BENHAMOU, F. Newton - constraint programming over nonlinear constraints. *Science of Computer Programming 30*, 1-2 (1998), 83–118.

[47] HOOKER, J. N. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. Wiley, 2000.

[48] HOOKER, J. N. A hybrid method for planning and scheduling. *Principles and Practice of Constraint Programming (CP 2004), Lecture Notes in Computer Science 3258* (2004), 305–316.

[49] HOOKER, J. N. A hybrid method for the planning and scheduling. *Constraints 10*, 4 (2005), 385–401.

[50] Hooker, J. N. *Integrated Methods for Optimization*. International Series in Operations Research & Management Science. Springer, 2007.

[51] ILOG S.A. *ILOG Solver User Manual*. ILOG, Inc., Mountain View, CA, 1996.

[52] ILOG S.A. *ILOG CPLEX 9.0 User Manual*. ILOG, Inc., Mountain View, CA, 2006.

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

[54] Jeroslow, R. G. *Logic-Based Decision Support: Mixed Integer Model Formulation*, vol. 40 of *Annals of Discrete Mathematics*. North-Holland, 1985.

[55] Jeroslow, R. G. Representability in mixed integer programming, i: characterization results. *Discrete Appl. Math. 17*, 3 (1987), 223–243.

[56] Junger, M., and Thienel, S. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization, 1998.

[57] Khachyan, L. G. A polynomial algorithm for linear programming. *Dokl. Akad. Mauk USSR*, 244 (1979), 373–395.

[58] Klee, V., and Minty, G. How good is the simplex algorithm. *Inequalities III* (1972), 159–172.

[59] Kouvelis, P., and Yu, G. *Robust Discrete Optimization and Its Applications*. Kluwer Academic Publishers, 1997.

[60] Land, A. H., and Doig, A. G. An automatic method of solving discrete programming problems. *Econometrica 28*, 3 (July 1960), 497–520.

[61] Letchford, A. N., and Lodi, A. An augment-and-branch-and-cut framework for mixed 0-1 programming. In *Combinatorial optimization - eureka, you shrink!* (New York, NY, USA, 2003), Springer-Verlag New York, Inc., pp. 119–133.

[62] Maravelias, C., and Grossmann, I. A hybrid MILP/CP decomposition approach for the continuous time scheduling of multipurpose batch plants. *Computers and Chemical Engineering 28* (2004).

[63] Marchand, H., Martin, A., Weismantel, R., and Wolsey, L. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics 123*, 1-3 (2002), 397–446.

[64] McAloon, K., Tretkoff, C., and Wetzel, G. Disjunctive programming and cooperative solvers. *n/a* (1998).

[65] Milano, M. *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer, 2005.

[66] Nemhauser, G. L., and Wolsey, L. A. A recursive procedure to generate all cuts for 0-1 mixed integer programs. *Mathematical Programming 46*, 3 (1990), 379–390.

[67] Owen, G. Cutting planes for programs with disjunctive constraints. *Journal of Optimization Theory Applications 11* (1973), 49–55.

[68] Rauch, Henzinger, M., and V., K. Maintaining minimum spanning trees in dynamic graphs. In *Proceedings of the 24th International Colloquim on Automata, Languages and Programming (ICALP 1997)* (1997), pp. 594–605.

[69] Régin, J.-C. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of 12th National Conference on AI (AAAI'94)* (Seattle, July 31 - August 4 1994), vol. 1, pp. 362–367.

[70] Schrijver, A. On cutting planes. *Annals of Discrete Mathematics 9* (1980), 291–296.

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

[72] TÜRKAY, M., AND GROSSMANN, I. E. Logic-based minlp algorithms for the optimal synthesis of process networks. *Computers and Chemical Engineering 20* (1996), 959–978.

[73] VAN HENTENRYCK, P., MCALLESTER, D., AND KAPUR, D. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis 34*, 2 (1997), 797–827.

[74] VU, X.-H., SILAGHI, M.-C., SAM-HAROUD, D., AND FALTINGS, B. Branch-and-prune search strategies for numerical constraint solving, 2005.

[75] WILLIAM D. HARVEY, M. L. G. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1* (Montréal, Québec, Canada, August 20-25 1995), C. S. Mellish, Ed., Morgan Kaufmann, 1995, pp. 607–615.

[76] YAMAN, H., KARASAN, O. E., AND PINAR, M. C. The robust spanning tree problem with interval data. *Operations Research Letters 29* (2001), 31–40.

[77] YOUNG, R. D. Hypercilindrically deduced cuts in zero-one integer programs. *Operations Research 19* (1971), 1393–1405.

Appendix A

A.1 Lift and project cut generator: algorithm details

The procedure for generating lift and project cuts, as outlined in the current document, deals only with binary variables. That is we only generate disjunctive cuts for fractional binary variables). The method can be extended to handle non-binary (general integer constrained) variables, but this is beyond the scope of this work. However, with this extension in mind, we shall refer to these variables as **constrained** variables, rather than just binary. Wherever the assumption that a variable is binary is made in the pseudocode, we say so in a comment. Since the procedure involves pivoting, which starts from an optimal simplex tableau and at each step ends up with a new tableau, we adopt the following notation: all objects that are related to the initial (optimal) solution of the LP relaxation are denoted as $\overline{x}$. The objects related to the subsequent (current) LP tableau (whether optimal or not, feasible or not) are denoted with $\tilde{x}$. This is a slight departure from the notation used by Balas and Perregaard[8], in the sense that they use $\tilde{a}_{ij}$ to refer to the coefficient for nonbasic variable j in row i of the *current* tableau. We use $\tilde{a}_{ij}$ instead (and reserve $\overline{a}_{ij}$, as well as $\overline{x}$ and $\overline{\mathcal{B}}$ for the case when we talk about the *optimal* LP tableau). We also use the term *slack* for both slack and surplus variables, but whenever the distinction is necessary, we point it out.

Algorithm **pivot_cuts** $(LP, rounds, max_pivots, strengthen)$

$rounds$	number of times we solve the LP and generate cuts for each fractional variable of the optimal solution
max_pivots	maximum number of pivots allowed in order to strengthen each cut
$strengthen$	if true, we apply disjunctive modularization (i.e. integrality strengthening) to row k after each pivot
$\tilde{\mathcal{B}}$	the current basis (which, sometimes, may be infeasible)
$(\tilde{x}, \tilde{s})$	the current solution (structural and slack)

```
1   for (1 ≤ t ≤ rounds) do
        /* Solve LP and let x̄ be the optimal solution */
2       (x̄, s̄, B̄, Ā) ← optimal_solution (LP)
        /* Generate optimal lift-and-project cuts for all fractional components of x̄ */
        /* NOTE! Since the new cuts could make the primal tableau infeasile, we must use dual-simplex from this point */
3       foreach (structural, constrained variable k : x̄_k is fractional) do
4           (π, π_0) ← create_lift_and_project_cut (k, x̄, s̄, B̄, Ā, strengthen)
            /* add the L&P cut derived from x_k ≤ 0 ∨ x_k ≥ 1 to the LP relaxation */
5           add_cut (π, π_0, LP)
6       end for
7   end for
```

Figure A.1: Algorithm for generating cuts via pivoting

Algorithm **create_lift_and_project_cut** $(k, \overline{x}, \overline{s}, \overline{\mathcal{B}}, \overline{\tilde{A}}, strengthen)$

1. $optimal \leftarrow false$ /* will be set to true when we have an optimal lift-and-project cut from row k */
2. $degeneracy \leftarrow false$ /* will be set to true if we need to perturb row k to avoid degeneracy in CGLP */
3. $(\tilde{\mathcal{B}}, \tilde{A}) \leftarrow (\overline{\mathcal{B}}, \overline{A})$ /* we start from the optimal LP tableau and pivot until we obtain the optimal cut from row k */
4. **if** $(strengthen)$ **then**
   - /* Apply disjunctive modularization to row k */
5. $(\phi_J, \phi_0) \leftarrow$ **create_strengthened_row** $(\tilde{A}, J, k)$
6. $\tilde{A} \leftarrow$ **add_LP_tableau_row** $(\tilde{A}, \phi_J, \phi_0)$ /* add the new row to the tableau */
   - **WARNING!** The index k has changed as a result of appending a new modularized row (we now have a new variable, y_k, on which we will work instead of x_k).
7. $k \leftarrow$ index of the newly added row /* this new row becomes our cut generation row! */
8. **end if**
9. $J \leftarrow \{$ indices of all non-basic variables in the **current** solution $\}$
10. $pivots \leftarrow 0$ /* We use this to limit the number of pivots (which will lead to a suboptimal cut) */
11. **while** (**not** $optimal$ **and** $pivots < max_pivots$) **do**
    - /* the cut is not yet an optimal lift-and-project cut */
    - /* Can we strengthen the cut (i.e. improve the solution of CGLP) by pivoting in the current LP tableau? */
12. **if** $(i^* \rightarrow$ ***find_cut_improving_pivot_row*** $(\tilde{A}, \tilde{\mathcal{B}}, J, k)$) **then**
13. $j^* \leftarrow$ **find_best_pivot_column** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, i^*, strengthen)$
14. $(\tilde{A}, \tilde{\mathcal{B}}, J) \leftarrow$ **change_basis** $(\tilde{A}, \tilde{\mathcal{B}}, J, i^*, j^*, strengthen)$
15. $pivots \leftarrow pivots + 1$
16. **else** /* in practice, this branch can be skipped, since the cut obtained without it has roughly the same strength */
17. **if** (row k has no 0 entries) **then**
18. $optimal \leftarrow true$ /* we are done, the cut is optimal */
19. **else** /* we must perturb row k such that the partition (M_1^i, M_2^i) (i.e. the basis in CGLP) becomes unique */
20. $degeneracy \leftarrow true$
21. $\tilde{A} \leftarrow$ **perturb_row** $(\tilde{A}, k)$
22. **end if**
23. **end if**
24. **end while**
25. **if** $(degeneracy)$ **then**
26. $\tilde{A} \leftarrow$ **remove_perturbation_from_row** $(\tilde{A}, k)$
27. **end if**
28. $(\pi, \pi_0) \leftarrow$ **create_intersection_cut** $(\tilde{A}, J, k)$ /* (π, π_0) is now a L&P cut */
29. $\pi \leftarrow$ **strengthen_cut** $(\tilde{A}, J, k, \pi)$ /* (π, π_0) is now a strengthened L&P (MIG) cut */
30. **return** (π, π_0)

Figure A.2: Algorithm for generating lift-and-project cuts

Procedure **find_cut_improving_pivot_row** $(\tilde{A}, \tilde{\mathcal{B}}, J, k)$

/* *choose a row i, some multiple of which will be added to row k* */

1 **foreach** (row $i \neq k$ of $\tilde{A}$) **do**

/* *First, assume x_i will go to its lower bound (i.e. replace $x'_i = UB_i - x_i$* */

2 **adjustTableauRow** (A, i, UB)

3 $\overline{r}_{u_i} \leftarrow$ **compute_CGLP_reduced_cost** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, \overline{x}, \overline{s}, i, 1)$

/* *Should be computed only if $r_{u_i} \geq 0$* */

4 $\overline{r}_{v_i} \leftarrow$ **compute_CGLP_reduced_cost** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, \overline{x}, \overline{s}, i, 2)$

/* *Now, assume x_i will go to its lower bound (i.e. replace $x'_i = x_i - LB_i$* */

5 **adjustTableauRow** (A, i, LB)

6 $\underline{r}_{u_i} \leftarrow$ **compute_CGLP_reduced_cost** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, \overline{x}, \overline{s}, i, 1)$

/* *Should be computed only if $r_{u_i} \geq 0$* */

7 $\underline{r}_{v_i} \leftarrow$ **compute_CGLP_reduced_cost** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, \overline{x}, \overline{s}, i, 2)$

8 **if** $(r_{u_i} < 0 \vee r_{v_i} < 0)$ **then**

/* *solution of CGLP (i.e. lift-and-project cut) can be improved* */

9 **return** i

/* *Pivoting on row i in LP (i.e. pivoting x_i out) corresponds to pivoting one of u_i or v_i into the basis of CGLP* */

10 **end if**

11 **end for**

/* *No pivoting row found!* */

12 **return** 0

Figure A.3: Finding a pivoting row (i.e. variable to be removed from $\tilde{\mathcal{B}}$)

Procedure **find_best_pivot_column** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, i, strengthen)$

We want to choose a column j to decide the sign and magnitude of the multiplier with which row i will be added to row k:
$$j_i^* \leftarrow \min_{j \in J \left| -\frac{\tilde{a}_{k0}}{\tilde{a}_{i0}} < \gamma_j = -\frac{\tilde{a}_{kj}}{\tilde{a}_{ij}} < \frac{1-\tilde{a}_{k0}}{\tilde{a}_{i0}} \right.} \left\{ \min_{j \in J | \gamma_j > 0} f_i^+(\gamma_j), \min_{j \in J | \gamma_j < 0} f_i^-(\gamma_j) \right\}$$

1 $m^* \leftarrow 0,\ j_i^* \leftarrow 0$ /* $j_i^* = 0$ means that column j_i^* is undefined */
2 **foreach** $(j \in J \left| -\frac{\tilde{a}_{k0}}{\tilde{a}_{i0}} < \gamma = \gamma_j = -\frac{\tilde{a}_{kj}}{\tilde{a}_{ij}} < \frac{1-\tilde{a}_{k0}}{\tilde{a}_{i0}} \right.)$ **do**
3 $m_j \leftarrow$ **CGLP_objective_function** $(A, J, i, j, \gamma, \overline{x}, \overline{s}, strengthen)$
4 **if** $(m_j < m^*)$ **then**
5   $m^* \leftarrow m_j,\ j_i^* \leftarrow j$
6 **end if**
7 **end for**
8 **return** j_i^*

Figure A.4: Finding the best pivoting column in the current basis $\tilde{\mathcal{B}}$

Procedure **change_basis** $(\tilde{A}, \tilde{\mathcal{B}}, J, i, l, strengthen)$

1 $(\tilde{A}, \tilde{\mathcal{B}}, J) \leftarrow$ **pivot** $(\tilde{A}, \tilde{\mathcal{B}}, J, i, l)$
2 **if** $(strengthen)$ **then**
 /* Apply disjunctive modularization to row k */
3 $(\phi_J, \phi_0) \leftarrow$ **create_strengthened_row** $(\tilde{A}, J, k)$ /* Replace row k with the newly computed row */
4 $\tilde{A} \leftarrow$ **remove_LP_tableau_row** $(\tilde{A}, k)$
5 $\tilde{A} \leftarrow$ **add_LP_tableau_row** $(\tilde{A}, \phi_J, \phi_0)$ /* add the new row to the tableau */
6 $k \leftarrow$ index of the newly added row /* this new row becomes our cut generation row! */
7 **end if**
8 **return** $(k, \tilde{A}, \tilde{\mathcal{B}}, J)$

Figure A.5: Changing the current basis $\tilde{\mathcal{B}}$ in the LP tableau

Procedure **CGLP_objective** $(\tilde{A}, J, i, l, \gamma, \overline{x}, \overline{s}, strengthen)$

If we pivot on row i and column l (i.e. $\gamma = -\frac{\tilde{a}_{kl}}{\tilde{a}_{il}}$) then:

New row k:

$\tilde{a}^{\gamma}_{ki} \leftarrow \gamma$
$\tilde{a}^{\gamma}_{kj} \leftarrow$ **new_row_coefficient** $(\tilde{a}_{kj}, \tilde{a}_{ij}, \gamma)$, for $j \in J \setminus \{ l \}$
$\tilde{a}^{\gamma}_{k0} \leftarrow$ **new_row_coefficient** $(\tilde{a}_{k0}, \tilde{a}_{i0}, \gamma)$

New **strengthened** row k (after modularization):

$\tilde{a}^{\gamma}_{ki} \leftarrow$ **strengthened_row_coefficient** $(\gamma, \tilde{a}^{\gamma}_{k0}, \textit{typeof}\ (\{ i \}, i)\ )$
$\tilde{a}^{\gamma}_{kj} \leftarrow$ **strengthened_row_coefficient** $(\tilde{a}^{\gamma}_{kj}, \tilde{a}^{\gamma}_{k0}, \textit{typeof}\ (J \setminus \{ l \}, j)\ )$, for $j \in J \setminus \{ l \}$
$\tilde{a}^{\gamma}_{k0} \leftarrow \tilde{a}^{\gamma}_{k0}$

IMPORTANT! Note that we apply disjunctive modularization to $\tilde{a}^{\gamma}_{kj}$ (i.e. the new row), NOT to $\tilde{a}_{kj}$ (i.e. the row before pivoting). This means that when we compute the CGLP objective to determine which column to pivot on, we must BOTH simulate the pivot (i.e. compute $\tilde{a}^{\gamma}_{kj}$) and apply disjunctive modularization to the new (*simulated*) row! Of course, the second step is only necessary when we use disjunctive modularization.

New simple disjunctive cut from new row k:

$\pi^{\gamma}_{i} \leftarrow$ **intersection_cut_coefficient** $(\gamma, \tilde{a}^{\gamma}_{k0})$
$\pi^{\gamma}_{j} \leftarrow$ **intersection_cut_coefficient** $(\tilde{a}^{\gamma}_{kj}, \tilde{a}^{\gamma}_{k0})$, for $j \in J \setminus \{ l \}$
$\pi^{\gamma}_{0} \leftarrow$ **intersection_cut_rhs** $(\tilde{a}^{\gamma}_{k0})$

New **strengthened** intersection cut from new row k:

$\pi^{\gamma}_{i} \leftarrow$ **strengthened_intersection_cut_coefficient** $(\gamma, \tilde{a}^{\gamma}_{k0}, \textit{typeof}\ (\{ i \}, i)\ )$
$\pi^{\gamma}_{j} \leftarrow$ **strengthened_intersection_cut_coefficient** $(\tilde{a}^{\gamma}_{kj}, \tilde{a}^{\gamma}_{k0}, \textit{typeof}\ (J \setminus \{ l \}, j)\ )$, for $j \in J \setminus \{ l \}$
$\pi^{\gamma}_{0} \leftarrow$ **intersection_cut_rhs** $(\tilde{a}^{\gamma}_{k0})$

Objective function of (CGLP)$_k$ corresponding to the cut $\pi^{\gamma}_{i} x_i + \pi^{\gamma}_{J} s_J \geq \pi^{\gamma}_{0}$:

$$f_{CGLP} = \frac{-\pi^{\gamma}_{0} + \sum_{j \in (J \setminus \{ l \}) \cup \{ i \}} \pi^{\gamma}_{j} s_j}{1 + \sum_{j \in (J \setminus \{ l \}) \cup \{ i \}} \left| \tilde{a}^{\gamma}_{kj} \right|}$$

IMPORTANT! Note that using $|\gamma|$ in the denominator as in [8] is **misleading** (and not always correct)! Thus, the formula:

$$\frac{\pi^{\gamma}_{i} x_i + \pi^{\gamma}_{J} s_J - \pi^{\gamma}_{0}}{1 + |\gamma| + \sum_{j \in J \setminus \{ l \}} |\tilde{a}^{\gamma}_{kj}|}$$

is not correct if we use disjunctive modularization, because γ stands for the value of $\tilde{a}^{\gamma}_{ki}$, which, when we use disjunctive modularization is no longer just γ! The correct formula is shown below:

1 **return** $\frac{-\pi^{\gamma}_{0} + \pi^{\gamma}_{i} x_i + \pi^{\gamma}_{J'} s_{J'}}{1 + \left|\tilde{a}^{\gamma}_{ki}\right| + \sum_{j \in J'} \left|\tilde{a}^{\gamma}_{kj}\right|}$, where $J' = J \setminus \{ l \}$

Figure A.6: Computing the CGLP objective given by a prospective pivot (γ)

Procedure **intersection_cut_coefficient** (α, β)

1 **return** $\max\{\alpha(1-\beta), -\alpha\beta\} = -\alpha\beta + \max\{\alpha, 0\}$

Figure A.7: Computing coefficients for intersection cuts

Procedure **strengthened_intersection_cut_coefficient** $(\alpha, \beta, vartype)$

/ Strengthen the coefficients that correspond to* **constrained** *(i.e. structural) non-basic variables */*

/ using the integrality conditions on these variables */*

1 **return** $\begin{cases} \min\{(\alpha - \lfloor\alpha\rfloor)(1-\beta), (\lceil\alpha\rceil - \alpha)\beta\} & vartype \text{ is } \textbf{constrained} \\ \textbf{\textit{intersection_cut_coefficient}}(\alpha, \beta) & vartype \text{ is continuous} \end{cases}$

Figure A.8: Computing strengthened coefficients for intersection cuts

Procedure **new_row_coefficient** $(\tilde{a}_{kj}, \tilde{a}_{ij}, \gamma, strengthen, vartype)$

> If we pivot on row i and column l (i.e. $\gamma = -\frac{\tilde{a}_{kl}}{\tilde{a}_{il}}$) then:
>
> New row k:
>
> $\tilde{a}^{\gamma}_{ki} \leftarrow \gamma$
>
> $\tilde{a}^{\gamma}_{kj} \leftarrow$ **new_row_coefficient** $(\tilde{a}_{kj}, \tilde{a}_{ij}, \gamma)$, for $j \in J \setminus \{l\}$
>
> $\tilde{a}^{\gamma}_{k0} \leftarrow$ **new_row_coefficient** $(\tilde{a}_{k0}, \tilde{a}_{i0}, \gamma)$
>
> New **strengthened** row k (after modularization):
>
> $\tilde{a}^{\gamma}_{ki} \leftarrow$ **strengthened_row_coefficient** $(\gamma, \tilde{a}^{\gamma}_{k0}, \textit{typeof}(\{i\}, i))$
>
> $\tilde{a}^{\gamma}_{kj} \leftarrow$ **strengthened_row_coefficient** $(\tilde{a}^{\gamma}_{kj}, \tilde{a}^{\gamma}_{k0}, \textit{typeof}(J \setminus \{l\}, j))$, for $j \in J \setminus \{l\}$
>
> $\tilde{a}^{\gamma}_{k0} \leftarrow \tilde{a}^{\gamma}_{k0}$

1 $\tilde{a}^{\gamma}_{kj} \leftarrow \tilde{a}_{kj} + \gamma a_{ij}$
2 **if** $(strengthen)$ **then**
3 $\quad \tilde{a}^{\gamma}_{kj} \leftarrow$ **strengthened_row_coefficient** $(\tilde{a}^{\gamma}_{kj}, \tilde{a}^{\gamma}_{k0}, vartype)$ */* Strengthen the new coefficient */*
4 **end if**
5 **return** $\tilde{a}^{\gamma}_{kj}$

Figure A.9: Computing the coefficient of a row after a prospective pivot (γ)

Procedure **pivot** $(\tilde{A}, \tilde{\mathcal{B}}, J, i, l)$

If we pivot on row i and column l (i.e. $\gamma = -\frac{\tilde{a}_{kl}}{\tilde{a}_{il}}$) then:

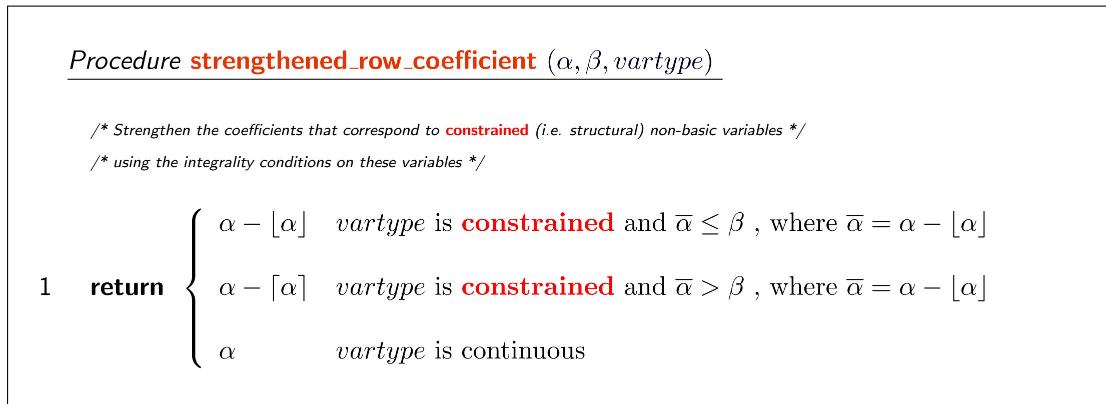
New row k:

$$\tilde{a}^{\gamma}_{kj} \leftarrow \tilde{a}_{kj} + \gamma \tilde{a}_{ij}$$
$$\tilde{a}^{\gamma}_{k0} \leftarrow \tilde{a}_{k0} + \gamma \tilde{a}_{i0}$$

```
1   foreach (row k ≠ i) do
        /* row k = row k + γ row i */
2       γ ← −ã_kl / ã_il
3       foreach (j ∈ J ∪ { l }) do
4           ã^γ_kj ← ã_kj + γ ã_ij
5       end for
6       ã^γ_k0 ← ã_k0 + γ ã_i0

7   end for
8   return B̃
```

Figure A.10: Pivoting in the $\tilde{\mathcal{B}}$ tableau



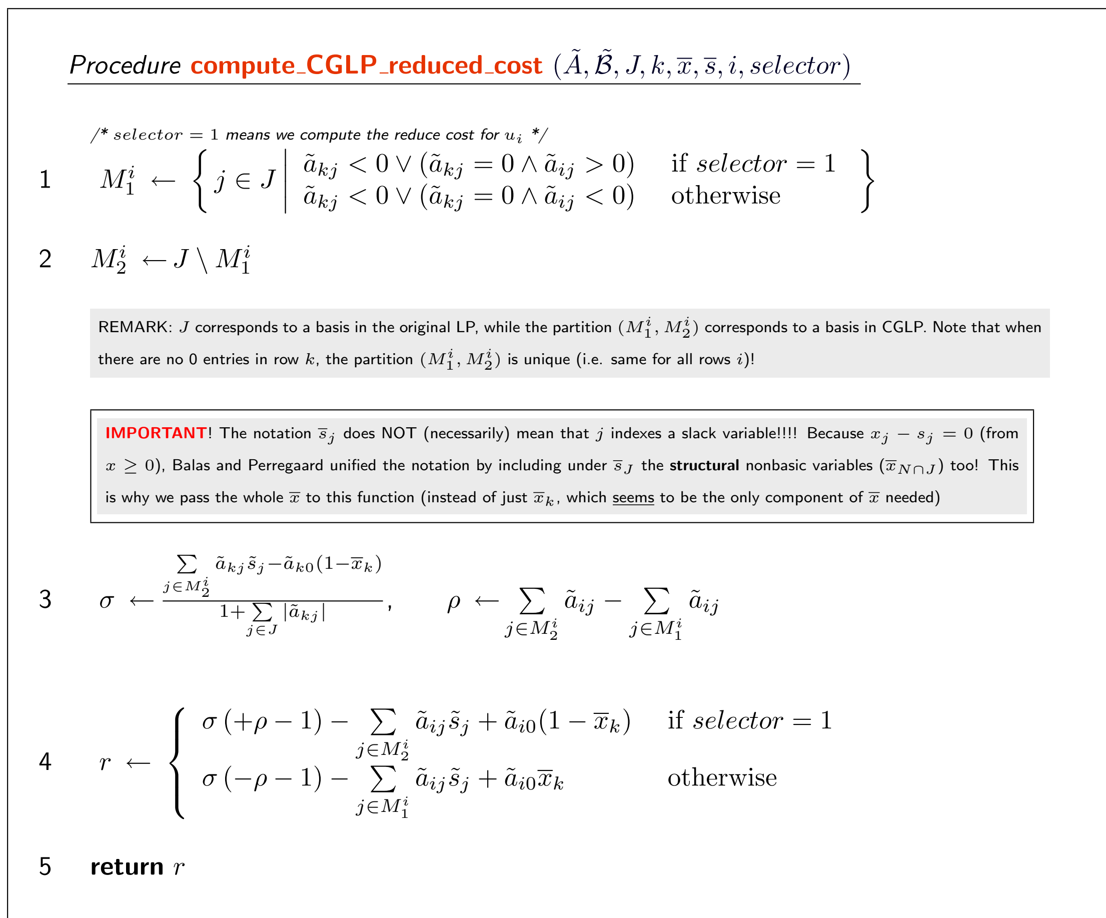
Procedure **strengthened_row_coefficient** $(\alpha, \beta, vartype)$

/ Strengthen the coefficients that correspond to* ***constrained*** *(i.e. structural) non-basic variables */*

/ using the integrality conditions on these variables */*

1 **return** $\begin{cases} \alpha - \lfloor \alpha \rfloor & vartype \text{ is } \textbf{constrained} \text{ and } \overline{\alpha} \le \beta \text{ , where } \overline{\alpha} = \alpha - \lfloor \alpha \rfloor \\ \alpha - \lceil \alpha \rceil & vartype \text{ is } \textbf{constrained} \text{ and } \overline{\alpha} > \beta \text{ , where } \overline{\alpha} = \alpha - \lfloor \alpha \rfloor \\ \alpha & vartype \text{ is continuous} \end{cases}$

Figure A.11: Computing row coefficients resulting from disjunctive modularization



Procedure **compute_CGLP_reduced_cost** $(\tilde{A}, \tilde{\mathcal{B}}, J, k, \overline{x}, \overline{s}, i, selector)$

/ $selector = 1$ means we compute the reduce cost for u_i */*

1 $M_1^i \leftarrow \left\{ j \in J \;\middle|\; \begin{matrix} \tilde{a}_{kj} < 0 \vee (\tilde{a}_{kj} = 0 \wedge \tilde{a}_{ij} > 0) & \text{if } selector = 1 \\ \tilde{a}_{kj} < 0 \vee (\tilde{a}_{kj} = 0 \wedge \tilde{a}_{ij} < 0) & \text{otherwise} \end{matrix} \right\}$

2 $M_2^i \leftarrow J \setminus M_1^i$

REMARK: J corresponds to a basis in the original LP, while the partition (M_1^i, M_2^i) corresponds to a basis in CGLP. Note that when there are no 0 entries in row k, the partition (M_1^i, M_2^i) is unique (i.e. same for all rows i)!

IMPORTANT! The notation $\overline{s}_j$ does NOT (necessarily) mean that j indexes a slack variable!!!! Because $x_j - s_j = 0$ (from $x \geq 0$), Balas and Perregaard unified the notation by including under $\overline{s}_J$ the **structural** nonbasic variables ($\overline{x}_{N \cap J}$) too! This is why we pass the whole $\overline{x}$ to this function (instead of just $\overline{x}_k$, which seems to be the only component of $\overline{x}$ needed)

3 $\sigma \leftarrow \frac{\sum_{j \in M_2^i} \tilde{a}_{kj} \tilde{s}_j - \tilde{a}_{k0}(1 - \overline{x}_k)}{1 + \sum_{j \in J} |\tilde{a}_{kj}|}, \qquad \rho \leftarrow \sum_{j \in M_2^i} \tilde{a}_{ij} - \sum_{j \in M_1^i} \tilde{a}_{ij}$

4 $r \leftarrow \begin{cases} \sigma(+\rho - 1) - \sum_{j \in M_2^i} \tilde{a}_{ij} \tilde{s}_j + \tilde{a}_{i0}(1 - \overline{x}_k) & \text{if } selector = 1 \\ \sigma(-\rho - 1) - \sum_{j \in M_1^i} \tilde{a}_{ij} \tilde{s}_j + \tilde{a}_{i0} \overline{x}_k & \text{otherwise} \end{cases}$

5 **return** r

Figure A.12: Compute CGLP reduced cost

Procedure **typeof** (J, j)

1 **return** $\begin{cases} \textbf{constrained} & j \in J \text{ and } x_j \text{ is } \textbf{constrained} \text{ (structural only)} \\ \text{continuous} & j \in J \text{ and } x_j \text{ is continuous (structural and slack/surplus)} \\ \text{ERROR} & j \notin J \end{cases}$

Figure A.13: Determining the type of a variable

Procedure **intersection_cut_rhs** (β)

1 **return** $\beta(1-\beta)$

Figure A.14: Computing rhs for intersection cuts

Procedure **create_intersection_cut** $(\tilde{A}, J, k)$

1 $\pi_0 \leftarrow$ **intersection_cut_rhs** $(\tilde{a}_{k0})$ /* $\pi_0 \leftarrow \tilde{a}_{k0}(1-\tilde{a}_{k0})$ */

2 $\pi_{j \in J} \leftarrow$ **intersection_cut_coefficient** $(\tilde{a}_{kj}, \tilde{a}_{k0})$ /* $\pi_{j \in J} \leftarrow \max\{\tilde{a}_{kj}(1-\tilde{a}_{k0}), -\tilde{a}_{kj}\tilde{a}_{k0}\}$ */

$\pi s_J \geq \pi_0$ is the **unstrengthened** simple disjunctive cut derived from the disjunction $x_k \leq 0 \vee x_k \geq 1$ applied to row k: $x_k = \tilde{a}_{k0} - \sum_{j \in J} \tilde{a}_{kj} s_j$

3 **return** (π, π_0)

Figure A.15: Intersection cut

Procedure **create_strengthed_row** $(\tilde{A}, J, k)$

1 $\phi_0 \leftarrow \tilde{a}_{k0}$

2 $\phi_{j \in J} \leftarrow$ **strengthed_row_coefficient** $(\tilde{a}_{kj}, \tilde{a}_{k0}, \textit{typeof}\,(J, j))$

3 **return** (ϕ, ϕ_0)

Figure A.16: Disjunctive modularization (integrality based strengthening)

Procedure **create_mixed_integer_gomory_cut** $(\tilde{A}, J, k)$

1 $(\pi, \pi_0) \leftarrow$ **create_intersection_cut** $(\tilde{A}, J, k)$
2 $\pi \leftarrow$ **strengthen_cut** $(\tilde{A}, J, k, \pi)$ /* *Strengthen the cut using the integrality conditions on* $x_{j \in J \setminus \{ k \}}$ */
3 **return** (π, π_0)

Figure A.17: Mixed integer Gomory cut

Procedure **strengthen_cut** $(\tilde{A}, J, k, \pi)$

/* *Strengthen the coefficients that correspond to* **constrained** *(i.e. structural) non-basic variables* */
/* *using the integrality conditions on these variables* */
1 $\pi_{j \in J \cap \{ 1, \ldots, p \}} \leftarrow$ **strengthened_intersection_cut_coefficient** $(\tilde{a}_{kj}, \tilde{a}_{k0},$ ***typeof*** $(J, j)\ )$
2 **return** π

Figure A.18: Cut strengthening using integrality conditions

Procedure **perturb_row** $(\tilde{A}, \tilde{\mathcal{B}}, k)$

1 $\tilde{a}_{kj} \leftarrow \epsilon^t,\ \forall\ j : \tilde{a}_{kj} = 0,\ t = 1, 2, \ldots$
2 **return** $\tilde{A}$

Figure A.19: Perturb a row of the $\tilde{\mathcal{B}}$ tableau

Procedure **remove_perturbation_from_row** $(\tilde{A}, \tilde{\mathcal{B}}, k)$

1 $\tilde{a}_{kj} \leftarrow 0,\ \forall\ j : \tilde{a}_{kj} = \epsilon^t,\ t = 1, 2, \ldots$
2 **return** $\tilde{A}$

Figure A.20: Remove perturbation from a row of the $\tilde{\mathcal{B}}$ tableau

Procedure **XPRESS_Get_Tableau_Row** (i)

Var	Size	Meaning
y	$rows$	temporary array to store $e_i B^{-1}$
z	$rows + cols + 1$	array in which we compute row i's coefficients (and right and side)
A^j	$rows$	temporary array in which we retrieve column j
b	$rows$	temporary array in which we retrieve the right hand side

```
1   y ← e_i                                                  /* Set up y = e_i to pick row i */
2   y ← XPRESSbtran (y)                                      /* y = e_i B^{-1} */

3   foreach (0 ≤ j ≤ rows − 1) do
        /* z = yA for each slack column of A */
4       z_j ← y_j
5   end for

6   foreach (rows ≤ j ≤ cols − 1) do
        /* z_{rows+j} = yA^j for each sructural column j of A */
7       A^j ← XPRESSgetcols (j)                              /* Retrieve column j from XPRESS */
8       z_{rows+j} ← yA^j
9   end for
                                  /* z_{rows+cols} = yb - compute right hand side of row i */
10  b ← XPRESSgetrhs ()                                      /* Retrieve RHS from XPRESS */
11  z_{rows+cols} ← yb
```

Important! We now need to bring all nonbasic variables at a lower/upper bound of zero, because all formulas in the paper assume this fact!

```
12  z ← XPRESS_adjust_row (z)

13  return z
```

Figure A.21: Retrieving the tableau from XPRESS

```
Procedure XPRESS_Get_var_bound (j)
```

Procedure **XPRESS_Get_var_bound** (j)

1 $v_j \leftarrow \begin{cases} UB_j & , j \text{ is structural, at its upper bound} \\ LB_j & , j \text{ is structural, at its lower bound} \\ & \\ RANGE_{row(j)} & , j \text{ is slack, at its upper bound} \\ 0 & , j \text{ is slack, at its lower bound} \end{cases}$

2 **return** v_j

Figure A.22: Function which returns the current bound of variable x_j

Procedure **XPRESS_adjust_row** (row)

/* row has $rows + cols$ elements, the last one being the RHS */

1 **foreach** $(0 \leq j \leq rows + cols - 1 | j \in J)$ **do**

/* bring each non-basic variable at a zero bound */

2 $bound_j \leftarrow$ **XPRESS_Get_var_bound** (j)

3 **if** $(var\ j\ is\ at\ UB)$ **then**

/* Var is at UB, replace with $x'_j = UB_j - x_j$ */

4 $row_j \leftarrow -row_j$

5 **else** /* Var is at LB, replace with $x'_j = x_j - LB_j$ */

6 **end if**

7 $row_{rows+cols} \leftarrow row_{rows+cols} + row_j bound_j$ /* adjust rhs to reflect the change of variable */

8 **end for**

9 **return** row

Figure A.23: Adjusting a row to reflect that all nonbasics are at a bound of zero

Procedure **XPRESS_Get_RHS** ()

We want to compute:

$$B^{-1}b - \sum_{j\in U} B^{-1}v_jA^j = B^{-1}\left(b - \sum_{j\in U} v_jA^j\right)$$

where U is the set of nonbasic variables at one of their bounds and v_j is the value of the bound at which variable j is (clearly, this only makes sense when variable j **is** at a bound (upper or lower) and that bound is non-zero.

```
1   b ← XPRESSgetrhs ()                                      /* Retrieve RHS from XPRESS */

    /* Compensate for non-basic variables with non-zero values */
2   U ← { j ∈ J | variable (or slack) j is at a (non-zero) bound }

3   foreach (j ∈ U) do
        /* look at all structural nonbasic variables at non-zero bounds */
4       bound_j ← XPRESS_Get_var_bound (j)

5       if (bound_j ≠ 0) then
            /* This test is here for efficiency, since a zero value would leave things unchanged! */
6           if (j is structural) then

7               A^j ← XPRESSgetcols (j)                      /* Retrieve column j from XPRESS */
8               b ← b − bound_j A^j                          /* Note that this is a vector operation! */
9           else                                             /* j is a slack, there is no column to retrieve */
                Note! There is no need (or possibility) to retrieve column j - it's the identity column with 1 in position i, where i is
                the row to which slack j corresponds. In fact, this column cannot even be retrieved from XPRESS - it is not stored
                explicitly (since we are dealing with columns of the original matrix, in which slacks don't exist)!
10              b_i ← b_i − bound_j, for i = row(j)
                /* Note! This is the same as above, with A^j = e_i, where i = row(j) */
11          end if
12      end if
13  end for

    /* We now have b − Σ_{j∈U} A^j, so we multiply this with B^{-1} */
14  b ← XPRESSftran (b)                                      /* rhs = B^{-1} (b − Σ_{j∈U} bound_j A^j) */

15  return b
```

Figure A.24: Computing the right hand side from XPRESS

Experimental results on cut modularization

The first experiment shows the effect of one round of cuts, with modularization applied *only* at the end of the pivoting sequence. In the second experiment, we apply modularization *after each pivot*. Both experiments were run with a limit of 50 cuts, and a maximum number of 20 pivots per cut. Tables A.2 (experiment 1) and A.4 (experiment 2) show the number of lift and project cuts generated for each problem, how many of them were actually optimal (i.e. we reached the CGLP minimum before exhausting the limit on the number of pivots), how many pivots we executed on average per cut, the density of the resulting cut (the number of non-zero coeffiecients in α), as well as the average and total cut generation time, in CPU seconds. Finally, the amount of gap closed by cuts at the root node is shown in Tables A.3 and A.5.

Problem name	Rows	Columns	Binary vars	Integer vars
10teams	231	2025	1799	0
a1c1s1	3313	3648	191	0
aflow30a	480	842	411	0
aflow40b	1443	2728	1358	0
air04	824	8904	8903	0
air05	427	7195	7194	0
arki001	1049	1388	414	123
cap6000	2177	6000	3974	0
danoint	665	521	55	0
disctom	400	10000	10000	0
ds	657	67732	67731	0
fast0507	508	63009	63008	0
fiber	364	1298	1240	0
fixnet6	479	878	377	0
gesa2	1393	1224	381	168
glass4	397	322	301	0
liu	2179	1156	1088	0
manna81	6481	3321	18	3303
markshare1	7	62	49	0
markshare2	8	74	59	0
mas74	14	151	148	0
mas76	13	151	130	0
misc07	213	260	258	0
mkc	3412	5325	5322	0
mod011	4481	10958	104	0
modglob	292	422	97	0
momentum2	24238	3732	1819	1
momentum3	56823	13532	6772	1
mzzv11	9500	10240	9988	251
mzzv42z	10461	11717	11479	235
net12	14022	14115	5946	0
noswot	183	128	74	25
nw04	37	87482	9078	0
opt1217	65	769	767	0
p2756	756	2756	2689	0
pk1	46	86	54	0
pp08a	137	240	63	0
pp08aCUTS	247	240	63	0
protfold	2113	1835	1834	0
qiu	1193	840	47	0
roll3000	2296	1166	243	492
rout	292	556	299	15
set1ch	493	712	234	0
seymour	4945	1372	1371	0
sp97ar	1762	14101	14100	0
stp3d	159489	204880	204880	0
swath	885	6805	6723	0
t1717	552	73885	73884	0
timtab1	172	397	63	107
timtab2	295	675	111	181
vpm2	235	378	333	0

Table A.1: Lift and project: properties of the problems in MIPLIB 2003

Problem name	L&P cuts	Optimal L&P cuts	Avg pivots per cut	Avg cut density	Avg cut time	Total cut time
10teams	50	0	20	0.94	0.197	9.85
a1c1s1	50	47	1	0.0008	0.396	19.8
aflow30a	31	8	15	0.0917	0.1061	3.29
aflow40b	41	2	19	0.0847	0.3134	12.85
air04	50	0	20	0.979	0.432	21.6
air05	50	0	20	0.985	1.18	59.1
arki001	46	19	11	0.0761	0.118	5.43
cap6000	2	0	20	0.999	0.595	1.19
danoint	35	0	20	0.85	0.0571	2
ds	50	0	20	0.53	5.723	286.1
fast0507	50	0	20	0.911	7.35	368
fiber	46	45	1	0.0424	0.198	9.11
fixnet6	50	40	4	0.027	0.053	2.65
gesa2	38	34	2	0.0067	0.155	5.88
glass4	50	50	0	0.0215	0.0224	1.12
liu	50	40	4	0.00311	0.092	4.6
manna81	2	2	0	0.0009	0.35	0.7
markshare1	6	5	4	0.903	0.00333	0.02
markshare2	7	4	10	0.905	0.00857	0.06
mas74	12	1	18	0.994	0.035	0.42
mas76	11	0	20	0.993	0.0182	0.2
misc07	20	7	14	0.611	0.0665	1.33
mkc	50	49	1	0.0272	1.42	71
mod011	18	8	11	0.0549	1.38	24.8
modglob	29	17	8	0.0343	0.0197	0.57
momentum2	50	27	9	0.0712	0.8538	42.69
mzzv11	50	6	17	0.0955	2.126	106.3
mzzv42z	50	35	8	0.0216	5.513	275.6
net12	50	34	7	0.0111	3.552	177.6
noswot	27	15	8	0.0573	0.00852	0.23
nw04	8	0	20	0.994	4.57	36.6
opt1217	28	1	19	0.238	0.09607	2.69
p2756	35	35	0	0.0128	0.176	6.15
pk1	15	6	13	0.814	0.00933	0.14
pp08a	50	43	2	0.0212	0.0112	0.56
pp08aCUTS	45	10	15	0.275	0.0122	0.55
protfold	50	0	20	0.923	0.2428	12.14
qiu	36	0	20	0.269	0.0514	1.85
roll3000	50	23	11	0.249	0.162	8.1
rout	31	4	18	0.307	0.0942	2.92
set1ch	50	50	0	0.003	0.0528	2.64
seymour	50	43	4	0.0854	0.747	37.4
sp97ar	50	5	18	0.873	2.699	135
swath	49	43	4	0.215	1.19	58.5
t1717	50	0	20	0.979	8.13	406.5
timtab1	50	40	4	0.0097	0.0214	1.07
timtab2	50	48	0	0.0062	0.0522	2.61
vpm2	50	44	3	0.0282	0.0408	2.04

Table A.2: Lift and project: properties of non-modularized cuts

Problem name	L&P cuts	LP bound before	LP bound after	Best known integer solution	Gap closed by cuts	Percentage of gap closed by cuts
10teams	50	917	917	924	0	0 %
air04	50	5.559e+04	5.561e+04	5.614e+04	26	4.71 %
air05	50	2.591e+04	2.593e+04	2.637e+04	20.47	4.44 %
arki001	46	7.58e+06	7.58e+06	7.581e+06	1.977	0.163 %
cap6000	2	-2.451e+06	-2.451e+06	-2.451e+06	27.91	30.8 %
danoint	35	62.69	62.69	65.67	0	0 %
fast0507	50	172.3	172.32	174	0.02	1.44 %
fiber	46	1.572e+05	2.721e+05	4.059e+05	1.149e+05	46.2 %
fixnet6	50	1326	1679	3983	353	13.3 %
gesa2	38	2.554e+07	2.561e+07	2.578e+07	7.663e+04	31.4 %
markshare1	6	0	0	1	0	0 %
markshare2	7	0	0	1	0	0 %
mas74	12	1.051e+04	1.058e+04	1.18e+04	65.18	5.05 %
misc07	20	1415	1425	2810	10	0.717 %
mkc	50	-605.2	-602.4	-553.8	2.8	5.58 %
mod011	18	-6.171e+07	-6.064e+07	-5.456e+07	1.067e+06	14.9 %
modglob	29	2.043e+07	2.049e+07	2.074e+07	5.231e+04	17.1 %
noswot	27	-43	-43	-43	0	0 %
nw04	8	1.631e+04	1.662e+04	1.686e+04	309.6	56.3 %
p2756	35	2699	2703	3124	4	0.875 %
pk1	15	0	0	11	0	0 %
pp08a	50	2981	5053	7350	2072	47.4 %
pp08aCUTS	45	5542	6109	7350	567	31.4 %
qiu	36	-931.6	-886	-132.9	45.6	5.72 %
rout	31	981.9	984.2	1078	2.3	2.41 %
set1ch	50	3.532e+04	3.833e+04	5.454e+04	3005	15.6 %
seymour	50	404.3	404.9	423	0.6	2.77 %
swath	49	334.6	374.8	497.6	40.2	24.7 %
vpm2	50	10.29	10.3	13.75	0.01	0.213 %

Table A.3: Lift and project: gap closed by non-modularized cuts

Problem name	L&P cuts	Optimal L&P cuts	Avg pivots per cut	Avg cut density	Avg cut time	Total cut time
10teams	50	0	20	0.951	0.14	7.02
a1c1s1	50	47	1	0.000828	0.3376	16.88
aflow30a	31	4	17	0.1	0.05935	1.84
air05	50	0	20	0.994	0.481	24.1
arki001	46	18	12	0.0861	0.111	5.1
cap6000	2	2	6	1	1.11	2.23
ds	50	0	20	0.531	3.689	184.4
fast0507	50	0	20	0.911	4.01	201
fiber	46	45	0	0.0537	0.109	5.02
fixnet6	50	40	4	0.027	0.0514	2.57
gesa2	38	31	3	0.00894	0.107	4.05
gesa2-o	38	33	2	0.00862	0.1103	4.19
glass4	50	50	0	0.0215	0.0208	1.04
liu	50	40	4	0.00346	0.1054	5.27
manna81	2	2	0	0.000903	0.355	0.71
markshare1	6	6	0	0.903	0.00167	0.01
markshare2	7	7	2	0.905	0.00429	0.03
misc07	20	8	14	0.651	0.0385	0.77
mod011	18	7	12	0.0549	1.15	20.6
modglob	29	17	8	0.0343	0.019	0.55
momentum1	50	24	10	0.0439	1.674	83.69
momentum2	50	27	9	0.0739	0.8624	43.12
msc98-ip	50	3	18	0.0488	2.298	114.9
noswot	27	15	8	0.0602	0.00593	0.16
nsrand-ipx	50	42	3	0.479	1.132	56.59
nw04	8	0	20	0.999	11.4	90.8
opt1217	28	0	20	0.524	0.06821	1.91
p2756	35	34	0	0.0135	0.248	8.67
pp08a	50	43	2	0.0212	0.0152	0.76
pp08aCUTS	45	11	15	0.276	0.0147	0.66
qiu	36	0	20	0.267	0.0417	1.5
rd-rplusc-21	50	19	12	0.0125	2.983	149.2
roll3000	50	22	11	0.254	0.2078	10.39
rout	31	1	19	0.385	0.0519	1.61
set1ch	50	50	0	0.00303	0.0674	3.37
seymour	50	37	6	0.158	0.99	49.5
t1717	50	4	18	0.991	15.93	796.3
timtab1	50	40	4	0.00972	0.0344	1.72
timtab2	50	48	0	0.00619	0.0758	3.79
tr12-30	50	50	0	0.00185	0.1018	5.09

Table A.4: Lift and project: properties of modularized cuts

Problem name	L&P cuts	LP bound before	LP bound after	Best known integer solution	Gap closed by cuts	Percentage of gap closed by cuts
10teams	50	917	917	924	0	0 %
air05	50	2.591e+04	2.593e+04	2.637e+04	17.41	3.78 %
arki001	46	7.58e+06	7.58e+06	7.581e+06	1.977	0.163 %
cap6000	2	-2.451e+06	-2.451e+06	-2.451e+06	16.38	18.1 %
fast0507	50	172.3	172.32	174	0.02	1.27 %
fiber	46	1.572e+05	2.311e+05	4.059e+05	7.388e+04	29.7 %
fixnet6	50	1326	1679	3983	353	13.3 %
gesa2	38	2.554e+07	2.561e+07	2.578e+07	7.363e+04	30.2 %
markshare1	6	0	0	1	0	0 %
markshare2	7	0	0	1	0	0 %
misc07	20	1415	1425	2810	10	0.717 %
mod011	18	-6.171e+07	-6.068e+07	-5.456e+07	1.028e+06	14.4 %
modglob	29	2.043e+07	2.049e+07	2.074e+07	5.235e+04	17.1 %
noswot	27	-43	-43	-43	0	0 %
nw04	8	1.631e+04	1.639e+04	1.686e+04	74.19	13.5 %
p2756	35	2699	2702	3124	3	0.73 %
pp08a	50	2981	5044	7350	2063	47.2 %
pp08aCUTS	45	5542	6103	7350	561	31 %
qiu	36	-931.6	-886.2	-132.9	45.4	5.69 %
rout	31	981.9	983.6	1078	1.7	1.77 %
set1ch	50	3.532e+04	3.833e+04	5.454e+04	3005	15.6 %
seymour	50	404.3	404.9	423	0.6	2.78 %

Table A.5: Lift and project: gap closed by modularized cuts