Logic-Based Methods for Optimization *

J. N. HOOKER

Graduate School of Industrial Administration

Carnegie Mellon University, Pittsburgh, PA 15213 USA

July 1994

Abstract

This tutorial describes a logic-based approach to formulating and solving pure and mixed integer programming problems. It develops logical counterparts for ideas associated with traditional branch-and-cut methods, such as cutting planes, facet-defining cuts, relaxations, etc. The motivations for doing this are a) to exploit the structure of a wide range of problems that are too complex for polyhedral analysis, b) to take advantage of logic processing techniques developed for constraint programming and logic programming, and c) to provide a unified approach to solving the growing number of problems with both qualitative and quantitative elements.

1 Introduction

The operations research community has traditionally regarded integer programming as linear programming with integrality constraints. This view is so well established that it seems odd to suggest that it could be anything else.

But an integer programming problem can be at least as naturally conceived as a problem in logical inference. The discrete element of a mixed integer programming problem can likewise

*Supported in part by Office of Naval Research Grant N00014-92-J-1028 and the Engineering Design Research Center at Carnegie Mellon University, funded by NSF grant 1-55093.

be cast in logical terms. The resulting formulation is at least as natural as the traditional one, and more general.

The traditional viewpoint has led to an emphasis on polyhedral methods and continuous relaxations. In many cases they are effective, but to focus so exclusively on this approach is unnecessarily limiting. The logical point of view opens the door to *logic-based methods*, which potentially offer several advantages.

- They provide an alternate way of thinking about combinatorial optimization that may suggest better solution methods.
- They can take advantage of recent progress in logical inference methods and forge links with constraint programming and logic programming.
- They provide a natural way to solve the growing number of models that contain both logical and quantitative elements.
- They allow one to extend to a more general setting many of the strategies for exploiting structure that operations researchers have developed.

This short tutorial will emphasize the last point. One of the great strengths of mathematical programming has been its ability to analyze deeply the structure of optimization models and use that structure to solve them. When the models are simple enough to permit this sort of analysis, powerful methods result. But practical models are often complex and messy. If a routing problem, for instance, can be posed as a traveling salesman problem, then the relevant polytope is well enough understood to permit optimal solution of large problems by adding strong cutting planes to a continuous relaxation. But if there are many vehicles of varying speeds and capacity, complex time windows, work rules, etc., then the polytope rapidly escapes our ability to analyze it, and useful cut generation becomes much harder.

The artificial intelligence community has tended to the opposite extreme. It has eschewed methods that presuppose structure in favor of generalized solution strategies with wide application. Logic-based methods have played an important role in this history, most obviously

in logic programming [28] (e.g., PROLOG [6, 36], PROLOG III [10]) but to some extent in constraint programming as well [37] (as represented by CHIP [11, 34], CHARME [7, 13, 32], PECOS, CHLEO, ConstraintLisp and other packages [2, 31]). But although these approaches are quite general, they are often incapable of obtaining good solutions because they do not adequately take advantage of the problem structure that does exist.

Logic provides one way of finding a middle ground—an approach that takes advantage of structure but can recognize a much broader array of structures than traditional mathematical programming. There is a specific reason for this: many of the concepts OR researchers have developed over the years to explore structure have logical analogs that permit some of the same strategies to be used in a more general context. Cutting planes, for instance, are nothing more than a special case of logical implications. The problem of generating cuts can be folded into the more general problem of drawing inferences, and inferential machinery developed over the years can be brought to bear. The notions of facet-defining and separating cuts, the most useful sort, have logical counterparts. Furthermore, the traditional continuous relaxations (linear, Lagrangean, etc.) can be replaced with discrete relaxations that have quite a different kind of structure, such as that of a partial k -tree. In fact, this more general perspective reveals that relaxation and cut generation are really two sides of the same coin and should be considered together.

Again, the motivating idea behind all of this is that one may be able to probe and take advantage of the logical structure of a problem even when the polytope is far too complex to analyze.

This exposition is more of a research proposal than a record of past successes. Logic-based methods are in their infancy and have yet to be vindicated as a general approach to optimization. But the initial efforts are successful enough, and the ideas enticing enough, to suggest that it may be worthwhile to explore these methods further.

A more extended treatment of this topic may be found in [21].

2 Historical Context

If logic-based methods for optimization are so attractive, why have they not gained acceptance already? Actually there is nothing new about them. Hammer and Rudeanu wrote a classic 1968 treatise [17] on boolean methods in operations research. Granot and Hammer [16] showed in 1971 how boolean methods might be used to solve integer programming problems.

Although boolean methods have seen applications (logical reduction techniques, solution of certain combinatorial problems), they have not been accepted as a general-purpose approach to optimization. There seem to be two main reasons for this. One is that they have not been demonstrated to be more effective than branch-and-cut. So there has been no apparent advantage in converting a problem to logical form.

But, again, it is too early to decide the issue. New methods and theoretical results are just coming to light, and bridges to constraint and logic programming are only beginning to be built.

A second reason for skepticism is that it is too hard to convert an inequality constraint to logical form. The most straightforward conversion is to write it as an equivalent set of logical propositions. But the number of propositions can grow exponentially with the number of variables in the inequality. Consider for instance the following constraint from a problem in Nemhauser and Wolsey ([29], p. 465).

$$\begin{aligned} &300x_3 + 300x_4 + 285x_5 + 285x_6 + 265x_8 + 265x_9 + 230x_{12} + 230x_{13} + 190x_{14} \\ &\quad + 200x_{22} + 400x_{23} + 200x_{24} + 400x_{25} + 200x_{26} + 400x_{27} + 200x_{28} + 400x_{29} \\ &\quad + 200x_{30} + 400x_{31} \leq 2700. \end{aligned} \tag{1}$$

Barth [3] reports that this constraint expands to 117,520 nonredundant logical clauses, using the method of Granot and Hammer [16].

This is true but irrelevant. It is foolish to expand an inequality constraint into its full logical equivalent. This is analogous to generating all possible cutting planes for an integer programming problem, which is never done. Practical algorithms generate a few “separating cuts,” and a closely analogous approach is available in the logical context.

3 Integer Programming as Logical Inference

Propositional logic, one of the simplest sorts of logic, concerns itself with formulas that are built up from simple (*atomic*) propositions. For instance, the proposition

$$(x_1 \vee x_2) \wedge \neg x_3$$

is true if and only if at least one of the atomic propositions x_1 and x_2 is true and x_3 is false. ($\wedge, \vee$ and $\neg$ mean ‘and,’ ‘or’ and ‘not,’ respectively.) In general a proposition F can be regarded as a boolean function $F(x_1, \dots, x_n) = F(x)$ of the truth values (1 for true, 0 for false) of atomic propositions x_j . F is true when $F(x) = 1$. One formula F *implies* another G when $G(x) \geq F(x)$ for all boolean x ; i.e., all 0-1 vectors x that make F true also make G true.

A 0-1 inequality $bx \geq \beta$ defines a boolean function $F(x)$ that is equal to 1 when x satisfies the inequality. (This is known as a *threshold function* in the electrical engineering literature [33].) The inequality can therefore be viewed as a logical proposition whose variables are atomic propositions. A system $Ax \geq a$ of 0-1 inequalities corresponds to the conjunction of the propositions in it. This means that $Ax \geq a$ implies an inequality $bx \geq \beta$, in the logical sense, when all 0-1 values x that satisfy the former also satisfy the latter. But this is precisely what a valid cut is. I will state this obvious but fundamental fact as a theorem.

Theorem 1 *An inequality is a valid cut (in the polyhedral sense) for a system of inequalities if and only if it is logically implied by the system.*

More generally, I will define a *logic cut* for a constraint set to be any proposition, inequality or otherwise, that is implied by it. Thus an inequality is a cutting plane for $Ax \geq a$ if and only if it is a logic cut.

It should now be obvious why integer programming is logical inference problem. The optimal value of the integer programming problem

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax \geq a \\ & x_j \in \{0, 1\}, \text{ all } j. \end{aligned} \tag{2}$$

is the largest β for which $cx \geq \beta$ is a valid cut for $Ax \geq a$. So,

Theorem 2 *The integer programming problem (2) is equivalent to the logical inference problem,*

$$\begin{aligned} \max \quad & \beta \\ \text{s.t.} \quad & Ax \geq a \text{ implies } cx \geq \beta \end{aligned} \tag{3}$$

Linear programming can likewise be viewed as a logical inference problem, but with a restricted form of inference. Consider the linear relaxation of (2), obtained by replacing $x_j \in \{0, 1\}$ with $0 \leq x_j \leq 1$. By duality, its optimal value is the largest β for which $cx \geq \beta$ is a nonnegative linear combination of $Ax \geq a$ and $0 \leq x_j \leq 1$ (all j). So, the dual LP is equivalent to the inference problem (3) if the implications of $Ax \geq a$ are taken to be only those that can be obtained by forming nonnegative linear combinations of $Ax \geq a$ and $0 \leq x_j \leq 1$.

This suggests that the logical problem (3), with unrestricted inference, can be regarded as a natural dual (the *logical dual*) of an integer programming problem (2). Logical methods are therefore in some sense dual to traditional methods.

4 An Integer Programming Example

Recent work [18] suggests that some of the best methods for solving inference (satisfiability) problems are branching methods that are analogous to branch-and-bound methods¹. I will therefore focus on logic-based approaches that use branching methods. This will also give me an opportunity to illustrate how the traditional theory and practice of branch-and-cut approaches can be transplanted to a logical setting.

Consider the integer programming problem,

$$\min \quad 3x_1 + 4x_2 + 2x_3 \tag{4}$$

$$\text{s.t.} \quad 2x_1 - x_2 + x_3 \geq 1 \tag{5}$$

¹A set S of formulas implies formula F if and only if $S \wedge \neg F$ is unsatisfiable

$$-x_1 + 3x_2 + 5x_3 \geq 3 \tag{6}$$

$$x_j \in \{0, 1\}$$

The minimum value is 2, which is the largest β for which the constraints logically imply $3x_1 + 4x_2 + 2x_3 \geq \beta$.

Constraint (5) is equivalent to the conjunction of the two logical clauses,

$$x_1 \vee \neg x_2 \tag{7}$$

$$x_1 \vee x_3$$

and (6) is equivalent to the conjunction of

$$\neg x_1 \vee x_3 \tag{8}$$

$$x_2 \vee x_3.$$

(A logical *clause* is a disjunction of *literals*, which are atomic propositions or their negations.)

Figure 1 shows a logic-based branch-and-cut tree for the problem. At the root node I solve a discrete relaxation of the problem. Here I will solve the simplest possible relaxation: just ignore the constraints and obtain the solution $x = (x_1, x_2, x_3) = 0$ with value 0. Next I generate two *separating* logic cuts, which are cuts that are violated by the solution of the current relaxation. Note that not all of the clauses (7)-(8) are generated. These new cuts become part of the discrete relaxation.

I next branch on x_1 , keeping in mind that the choice of branching rule is critical in satisfiability algorithms [23]. At the left successor node I may apply some sort of logic processing to the relaxation to draw further inferences. Here I use *unit resolution*, also called *forward chaining*. Since x_1 is fixed to 0, I eliminate x_1 from the clauses in the obvious way, which fixes $x_3 = 0$. I likewise eliminate x_3 and continue as long as possible. If I discover a contradiction I backtrack but otherwise solve what is left of the relaxation (in this case, no clauses remain). Since the solution of the relaxation is feasible in (5)-(6), I have an upper bound of 2 on the optimal solution value.

Continuing the search at the right successor node, I fathom because the value 5 of the relaxation is greater than the current upper bound. This completes the search.

Figure 1: Logic-based solution of a small integer programming problem.

5 A Generic Branch-and-Cut Algorithm

The solution method illustrated in the previous section has four elements.

Enumeration. Straightforward branching is the most natural form, but other schemes have been proposed in the AI literature that use such devices as backjumping, backmarking, “no goods,” etc. [14, 15, 26, 35].

Relaxation and cut generation. These are inseparable, because the relaxation is obtained by generating logic cuts. Cuts can be generated for constraint sets that contain logical propositions as well as 0-1 inequalities, which means that logic models and logico-mathematical models can be solved within the same framework as integer programming models. The cuts should be separating and strong, but they should result in a relaxation that is structured enough for fast solution. (The constraints of the traditional LP relaxation are in fact cuts, even if they are not normally regarded as such. When we use a separation algorithm to generate additional cuts, we are not concerned about whether they preserve the relaxation’s structure, because it remains an LP. But in the more general logic-based context, the structure of the relaxation must be considered explicitly.)

Logic processing. An inference or constraint propagation algorithm can be used to extract information from the logic cuts generated so far, perhaps the fact that they are unsatisfiable. For instance, solving the traditional LP relaxation would not detect the infeasibility of the 0-1 inequalities on the left below. But logic processing of logic cuts they imply (right) reveals the inconsistency.

$$\begin{array}{ll}
 x_1 + x_2 \geq 1 & x_1 \vee x_2 \\
 x_1 + (1 - x_2) \geq 1 & x_1 \vee \neg x_2 \\
 (1 - x_1) + x_2 \geq 1 & \neg x_1 \vee x_2 \\
 (1 - x_1) + (1 - x_2) \geq 1 & \neg x_1 \vee \neg x_2
 \end{array}$$

Here the link with constraint programming and AI is especially strong.

Figure 2:

```

Logic-Based Branch-and-Cut Algorithm.

  Set  $UB = \infty$ .

  Execute Branch( $\emptyset, 0$ ).

End.

Procedure Branch( $S, k$ )

  If  $k = 0$  then

    the optimal solution is the best found so far

    (infeasible if none found); stop.

  Apply a logic processing algorithm to  $S$ .

  If no contradiction is found then

    Find the minimum LB of  $cx$  subject to a relaxation of  $S$ .

    If  $LB < UB$  then:

      Generate separating logic cuts.

      Branch:

        Pick a literal  $L$  containing a variable

        that occurs in  $S$ .

        Perform Branch( $S \cup \{L\}, k + 1$ ).

        Perform Branch( $S \cup \{\neg L\}, k + 1$ ).

  End.

```

Bounding. This is the same sort of thing we do in branch-and-cut.

These elements are incorporated into the rudimentary logic-based branch-and-cut algorithm (essentially a specialized A* search) in Fig. 2. It solves the integer programming problem (2).

Problem structure is most readily exploited in the relaxation and cut generation phase. Deep logic cuts can reflect the logical structure of the problem. The relaxation can be designed to be as similar to the original problem as possible while retaining ease of solution.

6 Relaxations

The most straightforward logical relaxation of a 0-1 constraint set is simply a list of all separating logic cuts. The cuts could be logical clauses or some other convenient class of formulas. One attractive class are *extended* clauses, which have the form

At least k of the literals $L_1, \dots, L_m$ are true.

Such a relaxation is in general an NP-hard problem but not necessarily hard to solve. For instance, if one first converts the integer programming problem (2) to a feasibility problem, the relaxations are propositional satisfiability problems than can be attacked with some of the effective new methods surveyed in [19, 21, 18]. The conversion is accomplished by removing the objective function and adding an inequality $cx \leq \beta$ to the constraint set.

Another general sort of relaxation, explored in [22], is based on an underlying k -tree structure [1]. A k -tree is a graph whose vertices can be removed in an order for which each vertex, when removed, is adjacent to all and only the vertices of a k -clique. When the vertex is removed, all edges incident to it are removed. A *partial k -tree* is any subgraph of a k -tree.

Suppose the relaxation consists of a set of clauses or extended clauses. For each clause let the set of variables in it be a “clause set.” Partition the set of all variables into sets, each of which is the nonempty intersection of some collection of clause sets but properly contains no such intersection. Construct a *dependency graph* whose vertices correspond to sets of the partition. An edge connects two vertices when the variables in the corresponding sets occur in a common clause. If the graph is a partial k -tree, then the relaxation can be solved by nonserial dynamic programming [4, 8] in $O(2^{k+1})$ time. Thus if one can generate a set of separating cuts that have a partial k -tree structure for small k , the relaxation can be solved quickly. This is done in [23], and initial computational results are encouraging.

The relaxation can also exploit one’s knowledge of the problem structure. The constraints of the IP, for instance, may be equivalent to a set of clauses most of which are *Horn*—a common situation in knowledge bases. (A Horn clause is one that contains at most one positive literal.) A relaxation that considers only the Horn clauses could provide a good lower bound. If the

objective function coefficients are nonnegative, such a relaxation can be solved in linear time with a simplified version of unit resolution [12] (i.e., eliminate only variables fixed to 1, and set all unfixed variables to 0).

7 Strong Cuts

The logical analog of a facet-defining cut is a *prime cut*, which is defined with respect to a class C of logical propositions. A prime cut for a system $Ax \geq a$ of inequalities is a logic cut F that is equivalent to any cut in C that is implied by $Ax \geq a$ and implies F . It is a *prime inequality* if C is the set of all inequalities (with integer coefficients and right-hand side).

Useful logic cuts in practice need not and ordinarily would not be prime cuts. But an investigation of how prime cuts can in principle be generated provides insight into the nature of strong logic cuts.

A fundamental result of integer programming, due to Chvátal [9], says that a finite procedure generates all facet-defining inequalities (the strongest cutting planes) for a 0-1 system $Ax \geq a$. A parallel result can be proved for logic-based programming [20]. Let a *clausal inequality* have the form $ax \geq 1 + n(a)$, where each $a_j \in \{0, 1, -1\}$ and $n(a)$ is the sum of the negative components of a . For instance, the inequality $x_1 + (1 - x_2) \geq 1$, or $x_1 - x_2 \geq 0$, represents the logical clause $x_1 \vee \neg x_2$.

If exactly one variable x_j occurs posited in one of a pair of clauses and negated in the other, the *resolvent* of the pair is a clause containing all of their literals except x_j and $\neg x_j$, and similarly for clausal inequalities. Let a *diagonal sum* be defined as illustrated by the following example.

$$\begin{array}{rcl}
 x_1 + 5x_2 + 3x_3 + x_4 & \geq & 4 \\
 2x_1 + 4x_2 + 3x_3 + x_4 & \geq & 4 \\
 2x_1 + 5x_2 + 2x_3 + x_4 & \geq & 4 \\
 2x_1 + 5x_2 + 3x_3 & \geq & 4 \\
 2x_1 + 5x_2 + 3x_3 + x_4 & \geq & 5
 \end{array}$$

The fifth inequality is the diagonal sum of the first four. Note that the first four inequalities are identical except that the diagonal term is reduced by one. Also the right-hand side of the sum is increased by one.

A resolvent can be “generated” from a set S of inequalities if it is a resolvent of two clausal inequalities, each of which is implied by a single inequality of S . A diagonal sum is “generated” in a similar sense. Finally, let a set T of inequalities be *monotone* when T contains all clausal inequalities, and for any given inequality $ax \geq \beta + n(a)$ in T , T contains all inequalities $a'x \geq \beta' + n(a')$ such that $|a'| \leq |a|$ and $0 \leq \beta' \leq \beta$.

Theorem 3 *Let T be a monotone set of inequalities, and let S contain all resolvents and diagonal sums in T in that can be recursively generated from a feasible 0-1 system $Ax \geq a$, up to equivalence. Then every prime inequality for $Ax \geq a$ with respect to T is equivalent to some inequality in S .*

The *rank* of a logic cut (analogous to the Chvátal rank of a polyhedral cut) is the minimum number of iterations of this recursive procedure required to generate the cut.

8 Example: Matching Problems

It is generally believed that facet-defining cuts are the best possible cuts. But logic cuts can be stronger and therefore more useful.² A good illustration of this is a nonbipartite matching problem. The augmenting paths traditionally used in the best matching algorithms [29] in effect rely on logic cuts that strictly imply the less useful facet-defining inequalities (odd-set constraints) for the problem.

A matching problem is defined on an undirected graph (V, E) for which each edge in E is given a weight. The edges connect vertices that may be matched or paired, and a *matching* pairs some or all of the vertices. A matching can therefore be regarded as a set of edges, at most one of which touches any given vertex. The *weighted matching problem* is to find a

²This section represents joint work with Ajai Kapoor.

maximum weight matching; i.e., matching that maximizes the total weight of the edges used in the matching.

The matching problem can be written,

$$\max \sum_{e \in E} x_e \quad (9)$$

$$\begin{aligned} \text{s.t.} \quad & \sum_{e \in \delta(v)} x_e \leq 1, \text{ for } v \in V \\ & x_e \in \{0, 1\}, \text{ } e \in E, \end{aligned} \quad (10)$$

where $\delta(v)$ is the set of edges incident to v . x_e is 1 when e is part of the matching and 0 otherwise.

The convex hull of possible matchings has a particularly simple description. It is based on the fact that a matching for a graph (U, E) with an odd number of vertices can have at most $\frac{|U|}{2}$ edges. So the following *odd set constraints* are valid:

$$\sum_{e \in E(U)} x_e \leq \frac{|U|}{2}, \text{ all } U \subset V \text{ with } |U| \geq 3 \text{ and odd,} \quad (11)$$

where $E(U)$ contains the edges in the subgraph of (V, E) induced by U . In fact (10)-(11) define the convex hull of matchings.

For the purposes of logical analysis it is convenient to reverse the sense of the matching constraints (10) and odd set constraints (11) by replacing variables x_e with $y_e = 1 - x_e$, so that $y_e = 1$ when edge e is absent from the matching.

$$\sum_{e \in \delta(v)} y_e \geq |\delta(v)| - 1, \text{ for } v \in V \quad (12)$$

$$\begin{aligned} \sum_{e \in E(U)} y_e &\geq |E(U)| - \frac{|U|}{2}, \\ &\text{all } U \subset V \text{ with } |U| \geq 3 \text{ and odd.} \end{aligned} \quad (13)$$

The following is proved in [21].

Theorem 4 *An odd set constraint (13) for a matching problem is a logic cut of rank at most*

$$|E(U)| - \frac{|U|}{2} - 1.$$

Figure 3: A very small matching problem.

Odd set constraints are strictly implied by *augmenting path cuts*. Consider a matching problem on the simple graph of Fig. 8. The odd set constraints (facet-defining cuts) are simply the matching constraints $y_1 + y_2 \geq 1$ and $y_2 + y_3 \geq 1$. They are strictly implied by the augmenting path cut $y_1 + 2y_2 + y_3 \geq 2$, which says that either edge 2 is not in the matching or else edges 1 and 3 are not in the matching.

In general a path of odd length whose edges correspond to $y_{j_1}, \dots, y_{j_m}$ defines an augmenting path cut,

$$\begin{aligned} \frac{m-1}{2}y_{j_1} + \frac{m+1}{2}y_{j_2} + \frac{m-1}{2}y_{j_3} + \dots + \frac{m+1}{2}y_{j_{m-1}} \\ + \frac{m-1}{2}y_{j_m} \geq \frac{(m-1)(m+1)}{2}, \end{aligned}$$

which says that if the $(m+1)/2$ odd segments belong to a matching, then none of the $(m-1)/2$ even segments may belong to it, and vice-versa.

9 Mixed Integer Programming

From a logical point of view, the feasible set of a mixed integer programming (MIP) problem is a union of finitely many polyhedra. For any substitution of values for the integer variables, the constraint set describes one of the polyhedra in the union.

A set of logical propositions can also be used to indicate which polyhedra belong to the union, allowing the integer variables to be eliminated from the model. Solution is a matter of enumerating the polyhedra and finding the one that contains the best solution.

Consider a general mixed integer programming (MIP) problem,

$$\min \quad cx + dy \tag{14}$$

$$\text{s.t. } Ax + By \geq a$$

$$y_j \in \{0, 1\}, j = 1, \dots, n,$$

A 0-1 point y is *feasible* if (x, y) is feasible for some x . Each 0-1 value of y is associated with a polyhedron $\Pi(y)$ in x -space, namely the set of points satisfying (14) when y is so fixed. The feasible region can therefore be regarded as the union of $\Pi(y)$ over all feasible y .

To write an MIP in logical form, regard the y_j 's as atomic propositions.

$$\min \quad cx + dy \tag{15}$$

$$\text{s.t. } y \in Y$$

$$x \in \bigcup_{y \in Y} \Pi(y),$$

Here $y \in Y$ represents a set of logical propositions. (15) is actually more general than (14), due to a theorem of Jeroslow [25, 27]. It states that (15) can be written in the form (14) if and only if the polyhedra $\Pi(y)$ all have the same recession cone.

A pure integer programming problem is a special case of (15) in which the propositions in Y are 0-1 inequalities and there are no continuous variables x . The polyhedral sets $\Pi(y)$ are empty. This implies a conception of the relation between pure and mixed integer programming that is very different from the usual.

An MIP in form (15) can be solved by a branch-and-cut algorithm that enumerates linear programming constraint sets defining $\Pi(y)$'s, where the enumeration is controlled by the logical propositions $y \in Y$. The enumeration can be markedly accelerated by the use of an expanded sense of logic cuts that obtain in an MIP setting, namely a *nonvalid* logic cut. These may cut off feasible solutions but do not change the optimal solution.

10 An MIP Example

The following example appears in [24]. Suppose one wants to decide which of three processing units to install in the processing network of Fig. 3. The units are represented as boxes. Naturally one must install unit 3 if the network is to process anything, and one must install

Figure 4: A simple processing network.

units 1 or 2. Let's suppose in addition that units 1 and 2 should not both be installed. There is a variable cost associated with the flow through each unit, a fixed cost with building the unit, and revenue with the finished product. If x_j 's represent flows as indicated in Fig. 13 and y_j 's are 0-1 variables indicating which units are installed, the problem has the following MIP model.

$$\min \quad 3x_3 + 2.8x_5 - 9x_7 + 2x_1 + z_1 + z_2 + z_3 \quad (16)$$

$$\text{s.t.} \quad x_1 - x_2 - x_4 = 0 \quad (17)$$

$$x_6 - x_3 - x_5 = 0 \quad (18)$$

$$x_3 - 0.9x_2 = 0 \quad (19)$$

$$x_5 - 0.85x_4 = 0 \quad (20)$$

$$x_7 - 0.75x_6 = 0 \quad (21)$$

$$x_7 \leq 10 \quad (22)$$

$$x_3 - 30y_1 \leq 0 \quad (23)$$

$$x_5 - 30y_2 \leq 0 \quad (24)$$

$$x_7 - 50y_3 \leq 0 \quad (25)$$

$$y_1 + y_2 \leq 1 \quad (26)$$

$$z_1 = 14y_1 \quad (27)$$

Figure 5: Branch-and-bound solution of a small mixed integer programming problem.

Constraints (17)-(18) are flow balance constraints. (19)-(21) specify yields from the processing units. (22) bounds the output. (23)-(25) are “Big M” constraints that prohibit flow through a unit unless it is built. (27)-(29) define the fixed costs.

A conventional branch-and-bound tree for this problem appears in Fig. 4. Note that the optimal solution is to build none of the units.

11 Logic-Based Solution of an MIP

I will now illustrate how logic-based branch-and-bound can solve an MIP problem in logical form. It is convenient to suppose that the objective function of (15) is simply cx . This can be done by introducing a continuous variable z_j for each $d_j \neq 0$, letting the z_j have coefficient 1 in the objective function, and augmenting $\Pi(y)$ with the constraint $z_j = d_j$ whenever $y_j = 1$. The generic algorithm appears in Fig. 5.

The example of the previous section is put in logical form as follows. Note first that the objective function is already of the form cx . The set S of logical constraints is simply $\{\neg y_1 \vee \neg y_2\}$, which corresponds to constraint (26). The linear constraint set $\Pi(y)$ consists of constraints (17)-(22), nonnegativity constraints, and the following:

$$\begin{aligned} x_3 = 0 \quad \text{if } y_1 = 0, \quad z_1 = 14 \quad \text{if } y_1 = 1 \\ x_5 = 0 \quad \text{if } y_2 = 0, \quad z_2 = 14 \quad \text{if } y_2 = 1 \\ x_7 = 0 \quad \text{if } y_3 = 0, \quad z_3 = 14 \quad \text{if } y_3 = 1 \end{aligned} \tag{30}$$

Note that $\Pi(y)$ is defined even when some components of y are undetermined ($y_j = u$).

Before solving this example, it is useful to introduce in the next section some additional logic cuts.

12 Nonvalid Logic Cuts

In the context of mixed integer programming it is useful to define a more general sense of logic cut. Let the *graph* G for a mixed integer optimization problem (15) be the set

$$\{(cx + dy, x, y) \mid y \in Y, x \in \bigcup_{y \in Y} \Pi(y)\}.$$

The *epigraph* E is

$$\{(z, x, y) \mid (z', x, y) \in G \text{ for some } z' \leq z\}.$$

The *projection* of the epigraph onto the space of continuous variables is

$$\{(z, x) \mid (z, x, y) \in E \text{ for some } y\}.$$

Figure 6:

```

Logic-Based Branch-and-Cut Algorithm for MIP.

  Set  $UB = \infty$ ,  $y = (u, \dots, u)$  (where  $u$  =undetermined).
  Execute Branch( $\emptyset, y, 0$ ).

End.

Procedure Branch( $S, y, k$ )
  If  $k = 0$  then
    the optimal solution is the best found so far
    (infeasible if none found); stop.
  Apply a logic processing algorithm to  $S$ ,
    fixing some variables in  $y$  if possible.
  If no contradiction is found, then
    Find the minimum LB of  $cx$  subject to  $x \in \Pi(y)$ .
    If  $LB < UB$  then:
      Generate separating logic cuts.
      Branch:
        Pick a literal  $L$  containing a variable
          that occurs in  $S$ .
        Perform Branch( $S \cup \{L\}, y, k + 1$ ).
        Perform Branch( $S \cup \{\neg L\}, y, k + 1$ ).

End.

```

A *logic cut* in the extended sense is a constraint $y \in T$ that, when added to the constraint set of (15), results in the same projected epigraph. The cut is *valid* if

$$\bigcup_{y \in Y} \Pi(y) = \bigcup_{y \in Y \cap T} \Pi(y).$$

A cut can be nonvalid (i.e., cut off feasible values of y), but it never changes the value of the optimal solution.

Some nonvalid logic cuts can be generated for the example of the previous section as follows. Note that it makes no sense to consider a solution in which a unit is installed but carries no flow. Yet such solutions can and do occur in the branch-and-bound tree. Nodes 5 and 8 of Fig. 4 have LP solutions in which the installed unit carries no flow. Computational experience [24, 30] suggests that such superfluous nodes can be very numerous in a branch-and-bound tree.

This situation can be prevented by adding constraints that allow a unit to be installed only if a downstream unit is installed:

$$\neg y_1 \vee y_3 \tag{31}$$

$$\neg y_2 \vee y_3 \tag{32}$$

and only if at least one upstream unit is installed:

$$y_1 \vee y_2 \vee \neg y_3. \tag{33}$$

These are nonvalid logic cuts because they cut off feasible values of (y_1, y_2, y_3) . It is shown in [24] that they essentially exhaust the nonvalid logic cuts for such a problem.

Figure 2 displays the search tree for a logic-based solution of the example that uses (31)-(33). Note that the tree is smaller than the branch-and-bound tree of Fig. 4. The superfluous nodes 5 and 8, as well as other nodes, have been deleted.

Logic-based methods have been applied to MIP models of chemical processing network design problems [24]. Although these problems would be difficult to analyze polyhedrally, the simple intuition described above leads to the generation of effective logic cuts. They solve

large problems substantially more rapidly than a state-of-the-art MIP solver with preprocessor (OSL), and in some cases solve problems that OSL cannot solve. Logic cuts are also being applied to truss structure design problems with discrete bar sizes [5].

References

- [1] Arnborg, S., and A. Proskurowski, Characterization and recognition of partial k -trees, *SIAM Journal on Algebraic and Discrete Methods* **7** (1986) 305-314.
- [2] Banel, T., J.-Y. Cras, J.-M. Kerisit, S. Sciamma, Programming by constraints: technology or method? *Genie logiciel et systemes experts* **27** (1992) 84-88.
- [3] Barth, P., Linear 0-1 inequalities and extended clauses, manuscript, Max-Planck-Institut für Informatik, W-6600 Saarbrücken, Germany, ca. 1993.
- [4] Bertele, U., and F. Brioschi, *Nonserial Dynamic Programming*, Academic Press (New York, 1972).
- [5] Bollapragada, S., O. Ghattas and J. N. Hooker, Logic-based optimization of truss structure design, Carnegie Mellon University, in preparation.
- [6] Bratko, I., *PROLOG Programming for Artificial Intelligence, International Computer Science*, Addison-Wesley (1986).
- [7] BULL Corporation, *CHARME VI User's Guide and Reference Manual*, Artificial Intelligence Development Centre, BULL S.A. (France, 1990).
- [8] Chhajed, D., and T. Lowe, Solving structured multifacility location problems efficiently, to appear in *Management Science*.
- [9] Chvátal, V., Edmonds polytopes and a hierarchy of combinatorial problems, *Discrete Mathematics* **4** (1973) 305-337.

Figure 7: Logic-based solution of the problem with nonvalid logic cuts.

- [10] Colmerauer, A., An introduction to Prolog III, *Communications of the ACM* **33** (1990) 52-68.
- [11] Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Bertier, The constraint programming language CHIP, *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, December 1988.
- [12] Dowling, W. F. and Gallier, J.H. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Logic Programming* 1984 **1** (3) pp. 267-284.
- [13] Drexler, A., and C. Jordan, A comparison of logic and mixed-integer programming solvers for batch sequencing with sequence-dependent setups, to appear in *ORSA Journal on Computing*.
- [14] Freuder, E., and R. Wallace, Partial constraint satisfaction, *Artificial Intelligence* **52** (1992).
- [15] Ginsberg, M. L., Dynamic backtracking. *Journal of Artificial Intelligence Research* **1** (1993) 25-46.
- [16] Granot, F., and P. L. Hammer, On the use of boolean functions in 0-1 linear programming, *Methods of Operations Research* (1971) 154-184.
- [17] Hammer, P. L., and S. Rudeanu, *Boolean Methods in Operations Research and Related Areas*, Springer Verlag (Berlin, New York, 1968).
- [18] Harche, F., J. N. Hooker and G. L. Thompson, A computational study of satisfiability algorithms for propositional logic, to appear in *ORSA Journal on Computing*.
- [19] Hooker, J. N., A quantitative approach to logical Inference. *Decision Support Systems* 1988 **4** pp. 45-69.
- [20] Hooker, J. N., Generalized resolution for 0-1 inequalities, *Annals of Mathematics and AI* **6** (1992) 271-286.

- [21] Hooker, J. N., Logic-based methods for optimization: A tutorial, presented at ORSA Computer Science Technical Section meeting, Williamsburg, VA, USA, January 1994. Postscript file or hard copy available from the author.
- [22] Hooker, J. N., and N. R. Natraj, A logic-based method for integer programming, in preparation.
- [23] Hooker, J. N., and V. Vinay, Branching rules for satisfiability, working paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213 USA, 1994.
- [24] Hooker, J. N., H. Yan, I. E. Grossmann, and R. Raman, Logic cuts for processing networks with fixed costs. *Computers and Operations Research* **21** (1994) 265-279.
- [25] Jeroslow, R. E., Representability in mixed integer programming, I: Characterization results, *Discrete Applied Mathematics* **17** (1987) 223-243.
- [26] Jiang, Y., T. Richards and B. Richards, No-good backmarking with min-conflict repair in constraint satisfaction and optimization, Proceedings, Second Workshop on Principles and Practice of Constraint Programming (1994) 36-47.
- [27] Jeroslow, R. E., and J. K. Lowe, Modeling with integer variables, *Mathematical Programming Studies* **22** (1984) 167-184.
- [28] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag (New York 1984).
- [29] Newhauser, G. L., and L. A. Wolsey, *Integer and Combinatorial Optimization* (Wiley, 1988).
- [30] Raman, R., and I. E. Grossmann, Relation between MILP modeling and logical inference for chemical process synthesis, *Computers and Chemical Engineering* **15** (1991) 73-84.
- [31] Remy, C., Programming by constraints, *Micro Systemes* No. 104 (1990) 147-150.

- [32] Sciamma, D., J. Gay, A. Guillard, CHARME: A constraint oriented approach to scheduling and resource allocation, *Artificial Intelligence in the Pacific Rim*, Proceedings of the Pacific Rim International Conference on Artificial Intelligence, Nagoya, Japan (1990) 71-76.
- [33] Sheng, C.-L., *Threshold Logic*, Academic Press (New York, 1969).(New York, 1969).
- [34] Simonis, H., and M. Dincbas, Propositional calculus problems in CHIP, in F. Benhamou and A. Colmerauer, eds., *Constraint Logic Programming: Selected Research*, MIT Press (Cambridge, MA, 1993) 269-285.
- [35] Stallman, R. M., and G. J. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* **9** (1977) 135-196.
- [36] Sterling, L., and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press (Cambridge, MA, 1986).
- [37] Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, MIT Press (Cambridge, MA, 1988).