Logic-Based Modeling

J. N. Hooker

September 27, 2002

Abstract

Logic-based modeling can result in models that are more natural
and easier to debug. The addition of logical constraints to mixed inte-
ger programming need not sacrifice computational speed and can even
enhance it if the constraints are processed correctly. They should be
written or automatically reformulated so as to be as nearly consis-
tent or hyperarc consistent as possible. They should also be provided
a tight continuous relaxation. This chapter show how to accomplish
these goals for a number of logic-based constraints: formulas of propo-
sitional logic, cardinality clauses, 0-1 linear inequalities (viewed as log-
ical formulas), cardinality rules, and mixed logical/linear constraints.
It does the same for three global constraints that are popular in con-
straint programming systems: the all-different, element and cumulative
constraints.

Models cannot be considered apart from their clarity and computational
tractability. This becomes evident when one considers the purposes of mod-
eling. In the broadest sense, a model is a simply a description or graphic
representation of some phenomenon. But historically models have often
been written in a formal or quasi-formal language, for two major reasons:
(a) to elucidate the structure of the phenomenon by describing it in a precise
and limited vocabulary, and (b) to allow one to compute consequences of the
model. The classical transportaion model, for example, not only displays
the problem as a network that is easy to understand, but it puts the problem
in a form that can be solved very efficiently with the transportation simplex
algorithm.

Optimization modeling has generally emphasized ease of computation
more heavily than the clarity and explanatory value of the model. (The
transportation model is a happy exception that is strong on both counts.)

This is due in part to the fact that optimization, at least in the context of operations research, is often more interested in prescription than description. Practitioners who model a manufacturing plant, for example, typically want a solution that tells them how the plant should be run. Yet a succinct and natural model offers several advantages: it is easier to construct, easier to debug, and more conducive to understanding how the plant works.

This chapter explores the option of enriching mixed integer programming (MILP) models with logic-based constraints, in order to provide more natural and succinct expression of logical conditions. Due to formulation and solution techniques developed over the last several years, a modeling enrichment of this sort need not sacrifice computational tractability and can even enhance it.

One historical reason for the de-emphasis of perspicuous models in operations research has been the enormous influence of linear programming. Even though it uses a very small number of primitive terms, such as linear inequalities and equations, a linear programming model can formulate a remarkably wide range of problems. The linear format almost always allows fast solution, unless the model is truly huge. It also provides such analysis tools as reduced costs, shadow prices and sensitivity ranges. There is therefore a substantial reward for reducing a problem to linear inequalities and equations, even when this obscures the structure of the problem.

When one moves beyond linear models, however, there are less compelling reasons for sacrificing clarity in order to express a problem in a language with a small number of primitives. There is no framework for discrete or discrete/continuous models, for example, that offers the advantages of linear programming. The mathematical programming community has long used MILP for this purpose, but MILP solvers are not nearly as robust as linear programming solvers, as one would expect because MILP formulates NP-hard problems. Reletaively small and innocent-looking problems can exceed the capabilities of any existing solver, such as the market sharing problems identified by Williams [32] and studied by Cornuejols and Dawande [15]. Even tractable problems may be soluble only when carefully formulated to obtain a tight linear relaxation or an effective branching scheme.

In addition MILP often forces logical conditions to be expressed in an unnatural way, perhaps using big-M constraints and the like. The formulation may be even less natural if one is to obtain a tight linear relaxation. Current solution technology requires that the traveling salesman problem be written with exponentially many constraints in order to represent a simple

all-different condition. MILP may provide no usable formulation at all for important problem classes, including some resource-constrained scheduling problems.

It is true that MILP has the advantage of a unified solution approach, since a single branch-and-cut solver can be applied to any MILP model one might write. Yet the introduction of logic-based and other higher-level constraints no longer sacrifices this advantage.

The discussion begins in Section 1 with a brief description of how solvers can accommodate logic-based constraints: namely, by automatically converting them to MILP constraints, or by designing a solver that integrates MILP and constraint programming. Section 2 describes what constitutes a good formulation for MILP and for an integrated solver. The remaining sections describe good formulations for each type of constraint: formulas of propositional logic, cardinality clauses, 0-1 linear inequalities (viewed as logical formulas), cardinality rules, and mixed logical/linear constraints. Section 8 briefly discusses three global constraints that are popular in constraint programming systems: the all-different, element and cumulative constraints. They are not purely logical constraints, but they illustrate how logical expressions are a special case of a more general approach to modeling that offers a variety of constraint types. The paper ends with a summary.

1 Solvers for Logic-Based Constraints

1.1 Two Approaches

There are at least two ways to deal with logic-based constraints in a unified solver.

- One possibility is to provide automatic reformulation of logic-based constraints into MILP constraints, and then to apply a standard MILP solver [25]. The reformulation can be designed to result in a tight linear relaxation. This is a viable approach, although it obscures some of the structure of the problem. There are specialized inference algorithms and relaxations (discussed below) that allow for efficient processing of logic-based constraints. Their existence argues for dealing directly with the logic-based idiom.

- A second approach is to design a unified solution method for a diversity of constraint types, using the methods of constraint programming

3

(CP). In fact, CP can be integrated with MILP to obtain hybrid solvers that accept CP's diversity of constraints and apply solution techniques from both CP and MILP. Hybrid approaches have advantages of their own, since they enable more elegant models as well as faster solution of many problems. Surveys of the relevant literature on hybrid solvers may be found in [11, 17, 20, 21, 24, 33].

Whether one uses automatic translation or a CP/MILP hybrid approach, constraints must be written or automatically reformulated with the algorithmic implications in mind. A good formulation for MILP is one with a tight linear relaxation. A good formulation for CP is as nearly "consistent" as possible. A consistent constraint set is defined rigorously below, but it is roughly analogous to a convex hull relaxation in MILP. A good formulation for a hybrid approach should be good for both MILP and CP whenever possible, but the strength of a hybrid approach is that it can benefit from a formulation that is good in either sense.

1.2   Structured Groups of Constraints

Very often, the structure of a problem is not adequately exploited unless constraints are considered in groups. A group of constraints can generally be given an MILP translation that is tighter than the combined translations of the individual constraints. The consistency-maintenance algorithms of CP are more effective when applied to a group of constraints whose overall structure can be exploited.

Structured groups can be identified and processed in three ways.

Automatic detection. Design solvers to detect special structure and process it appropriately, as is commonly done for network constraints in MILP solvers. However, since modelers are generally aware of the problem structure, it seems more efficient to obtain this information from them rather than expecting the solver to rediscover it.

Hand coding. Ask modelers to exploit structured constraint groups by hand. They can write a good MILP formulation for a group, or they can write a consistent formulation.

Structural labeling. Let modelers label specially structured constraint groups so that the solver can process them accordingly. The CP community implements this approach with the concept of a *global* constraint, which represents a structured set of more elementary constraints.

As an example of the third approach, the global constraint all-different(a, b, c) requires that a, b and c take distinct values and thus represents three inequations $a \neq b$, $a \neq c$ and $b \neq c$. The elementary constraints are specified by passing parameters to the global constraint, in this case the variables involved. Alternatively, the parameters could simply be a list of the constraints in the group. For instance, a global constraint cnf$(a \vee b, a \vee \neg b)$ can alert the solver that its arguments are propositional formulas in conjunctive normal form (defined below). This allows the solver to process the constraints with specialized inference algorithms.

2 Good Formulations

2.1 Tight Relaxations

A good formulation of an MILP model should have a tight linear relaxation. The tightest possible formulation is a *convex hull formulation*, whose continuous relaxation describes the convex hull of the model's feasible set. The convex hull is the intersection of all half planes containing the feasible set. At a minimum it contains inequalities that define all the facets of the convex hull and equations that define the affine hull (the smallest dimensional hyperplane that contains the feasible set). Such a formulation is ideal because it allows one to find an optimal solution by solving the linear programming relaxation.

Since there may be a large number of facet-defining inequalities, it is common in practice to generate *separating* inequalities that are violated by the optimal solution of the current relaxation. This must be done during the solution process, however, since at the modeling stage one does not know what solutions will be obtained from the relaxation. Fortunately, some common constraint types may have a convex hull relaxation that is simple enough to analyze and describe in advance. Note, however, that even when one writes a convex hull formulation of each constraint or each structured subset of constraints, the model as a whole is generally not a convex hull formulation.

It is unclear how to measure the tightness of a relaxation that does not describe the convex hull. In practice, a "tight" relaxation is simply one that provides a relatively good bound for a problems that are commonly solved.

2.2 Consistent Formulations

A good formulation for CP is *consistent*, meaning that its constraints explicitly rule out assignments of values to variables that cannot be part of a feasible solution. Note that consistency is not the same as satisfiability, as the term might suggest.

To make the idea more precise, suppose that constraint set S contains variables $x_1, \ldots, x_n$. Each variable x_j has a domain D_j, which is the initial set of values the variable may take (perhaps the reals, integers, etc.) Let a *partial assignment* (known as a *compound label* in the constraints community) specify values for some subset of the variables. Thus a partial assignment has the form

$$(x_{j_1}, \ldots, x_{j_k}) = (v_{j_1}, \ldots, v_{j_k}), \text{ where each } j_\ell \in D_{j_\ell} \tag{1}$$

A partial assignment (1) is *redundant* for S if it violates no constraint in S but cannot be extended to a feasible solution of S. That is, every complete assignment

$$(x_1, \ldots, x_n) = (v_1, \ldots, v_n), \text{ where each } j \in D_j$$

that is consistent with (1) violates some constraint in S. By convention, a partial assignment violates a constraint C only if it assigns some value to every variable in C. Thus if $D_1 = D_2 = \mathcal{R}$, the assignment $x_1 = -1$ does not violate the constraint $x_1 + x_2^2 \geq 0$ since x_2 has not been assigned a value. The assignment is redundant, however, since there is no feasible solution in which $x_1 = -1$. A constraint set S is *consistent* if there are no redundant assignments. Thus the constraint set $\{x_1 + x_2^2 \geq 0\}$ is not consistent.

It is easy to see that a consistent constraint set S can be solved without backtracking. First assign x_1 a value $v_1 \in D_1$ that violates no constraints in S. If there is no such value, S is unsatisfiable. Now assign x_2 a value in $v_2 \in D_2$ such that $(x_1, x_2) = (v_1, v_2)$ violates no constraints in S. Consistency guarantees that such a value exists. Continue in this fashion until a feasible solution is constructed. The key to this greedy algorithm is that can easily recognize a redundant partial assignment by checking whether it violates some individual constraint.

The greedy algorithm can be viewed as a branching algorithm that branches on variables in the order $x_1, \ldots, x_n$. Consistency ensures that no backtracking is required, whatever the branching order. It is this sense that a consistent constraint set is analogous to a convex hull formulation. Either

allows one to find a feasible solution without backtracking, in the former case by using the greedy algorithm just described, and in the latter case by solving the linear programming relaxation. However, just as a convex hull formulation for each constraint does not ensure a convex hull formulation for the whole problem, an analogous fact holds for consistency. Achieving consistency for each constraint or several subsets of constraints need not achieve consistency for the entire constraint set. Nonetheless the amount of backtracking tends to be less when some form of consistency is achieved for subsets of constraints.

Consistency maintenance is the process of achieving consistency, which is ordinarily done by adding new constraints to S, much as valid inequalities are generated in MILP to approximate a convex hull formulation. One difference between CP and MILP, however, is that they generate different types of constraints. Just as MILP limits itself to linear inequalities, CP ordinarily generates only certain kinds of constraints that are easy to process, primarily *in-domain* constraints that have the form $x_j \in \bar{D}_j$. Here $\bar{D}_j$ is a reduced domain for x_j, obtained by deleting some values from D_j that cannot be part of a feasible solution. Generation of linear inequalities can in principle build a convex hull relaxation, but unfortunately generation of in-domain constraints cannot in general build a consistent constraint set (although it can achieve hyperarc consistency, defined below). Since generating in-domain constraints is equivalent to reducing the domains D_j, consistency maintenance is often described as *domain reduction*.

Hyperarc consistency (also called *generalized arc consistency*) is generally maintained for variables with finite domains. It implies that no assignment to a single variable is redundant. That is, given any variable x_j, no assignment $x_j = v$ for $v \in D_j$ is redundant. Hyperarc consistency is obtained by generating all valid in-domain constraints and therefore achieves maximum domain reduction. It can be viewed as computing the projection of the feasible set onto each variable. A wide variety of "filtering" algorithms have been developed to maintain hyperarc consistency for particular types of constraints.

Bounds consistency is defined when the system maintains *interval domains* $[a_j, b_j]$ for numerical variables x_j. A constraint set S is bounds consistent if neither $x_j = a_j$ nor $x_j = b_j$ is redundant for any j. Thus bounds consistency achieves the narrowest interval domains, just as hyperarc consistency achieves the smallest finite domains. Bounds consistency is generally maintained by interval arithmetic and by specialized algorithms for global constraints with numerical variables.

7

Both hyperarc and bounds consistency tend to reduce backtracking, because CP systems typically branch by splitting a domain. If the domain is small or narrow, less splitting is required to find a feasible solution. Small and narrow domains also make domain reduction more effective as one descends into the search tree.

2.3 Prime Implications

Backtracking can be avoided by recognizing redundant partial assignments. In consistent constraint sets, redundant assignments can be recognized by checking whether they violate some individual constraint. Yet there is a weaker property than consistency that may provide easy recognition of redundant assignments: namely, the constraint set contains all of its prime implications.

Prime implications, roughly speaking, are the strongest constraints that can be inferred from a constraint set. A partial assignment can be checked for redundancy by checking whether it is redundant for individual prime implications, which may be much easier than checking whether it is redundant for an entire constraint set. For example, it is easy to check that the partial assignment $x_1 = -1$ is redundant for $x_1 + x_2^2 \geq 0$ even though it does not violate the constraint. In practice a solver would detect redundancy by applying a domain reduction algorithm to the individual prime implications.

In some cases deriving all prime implications achieves consistency. In such cases a redundant partial assignment is not only redundant for a prime implication but actually violates it. This occurs in propositional logic, for example.

To define the concept of prime implication, suppose constraints C and D contain variables in $x = (x_1, \ldots, x_n)$. C *implies* constraint D if all assignments to x that satisfy C also satisfy D. Constraints C and D are *equivalent* if they imply each other.

Let H be a finite class of constraints with variables in x. For instance, H might be a set of logical clauses, or cardinality clauses, or 0-1 inequalities (defined in subsequent sections). Let an H-*implication* of a constraint set S be a constraint in H that S implies. Constraint C is a *prime H-implication* of S if C is a H-implication of S, and every H-implication of S that implies C is equivalent to C. The following is easy to show.

Lemma 1 *Every H-implication of a constraint set S is implied by some prime H-implication of S.*

The next section states precisely how Lemma 1 allows one to recognize redundant partial assignments.

3   Propositional Logic

3.1   Basic Ideas

A *formal logic* is a language in which deductions are made solely on the basis of the form of statements, without regard to their specific meaning. In *propositional logic*, a statement is made up of *atomic propositions* that can be true or false. The form of the statement is given by how the atomic propositions are joined or modified by such logical expressions as *not* ($\neg$), *and* ($\wedge$), and inclusive *or* ($\vee$).

A proposition defines a *propositional function* that maps the truth values of its atomic propositions to the truth value of the whole proposition. For example, the proposition

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

contains atomic propositions a, b and defines a function f given by the following *truth table*:

a	b	$f(a,b)$
0	0	1
0	1	0
1	0	0
1	1	1

(2)

Here 0 and 1 denote *false* and *true* respectively. One can define additional symbols for implication ($\rightarrow$), equivalence ($\equiv$), and so forth. For any pair of formulas A, B

$$A \rightarrow B \quad =_{\text{def}} \quad \neg A \vee B$$
$$A \equiv B \quad =_{\text{def}} \quad (A \rightarrow B) \wedge (B \rightarrow A)$$

Note that (2) is the truth table for equivalence. A *tautology* is a formula whose propositional function is identically true.

Many complex logical conditions can be naturally rendered in logical

form. For example,

$$
\begin{array}{ll}
\text{Alice will go to the party only} & a \to c \\
\quad \text{if Charles goes.} & \\
\text{Betty will not go to the party if} & (a \lor d) \to \neg b \\
\quad \text{Alice or Diane goes.} & \\
\text{Charles will go to the party} & c \to (\neg d \land \neg e) \\
\quad \text{unless Diane or Edward goes.} & \\
\text{Diane will not go to the party} & d \to (c \lor e) \\
\quad \text{unless Charles or Edward goes.} & \\
\text{Betty and Charles never go to the} & \neg(b \land c) \\
\quad \text{same party.} & \\
\text{Betty and Edward are always} & b \equiv e \\
\quad \text{seen together.} & \\
\text{Charles goes to every party that} & b \to c \\
\quad \text{Betty goes to.} &
\end{array}
\tag{3}
$$

3.2 Conversion to Clausal Form

It may be useful to convert formulas to clausal form, particularly since the well-known resolution algorithm is designed for clauses, and clauses have an obvious MILP representation.

Clausal form may be defined as follows. A *literal* has the form a or $\neg a$, where a is an atomic proposition. A *clause* is a disjunction of zero or more literals. (A clause with zero literals is regarded as necessarily false.) A formula is in *clausal form*, also known as *conjunctive normal form* (CNF), if it is a conjunction of one or more clauses.

Any propositional formula can be converted to clausal form in at least two ways. The more straightforward conversion requires exponential space in the worst case. It is accomplished by applying some elementary logical equivalences.

$$
\begin{array}{lrcl}
\textit{De Morgan's laws} & \neg(F \land G) & \equiv & \neg F \lor \neg G \\
& \neg(F \lor G) & \equiv & \neg F \land \neg G \\
\\
\textit{Distribution laws} & F \land (G \lor H) & \equiv & (F \land G) \lor (F \land H) \\
& F \lor (G \land H) & \equiv & (F \lor G) \land (F \lor H) \\
\\
\textit{Double negation} & \neg\neg F & \equiv & F
\end{array}
$$

```
Let F be the formula to be converted.
Set  k = 0 and S = ∅.
Replace all subformulas of the form G ≡ H with (G → H) ∧ (H → G).
Replace all subformulas of the form G ⊃ H with ¬G ∨ H.
Perform Convert(F).
The CNF form is the conjunction of clauses in S.

Function Convert(F)
   If F is a clause then add F to S.
   Else if F has the form ¬¬G then perform Convert(G).
   Else if F has the form G ∧ H then
      perform Convert(G) and Convert(H).
   Else if F has the form ¬(G ∧ H) then perform Convert(¬G ∨ ¬H).
   Else if F has the form ¬(G ∨ H) then perform Convert(¬G ∧ ¬H).
   Else if F has the form G ∨ (H ∧ I) then
      Perform Convert(G ∨ H) and Convert(G ∨ I).
```

Figure 1: *Conversion of F to CNF without additional variables. A formula of the form $(H \wedge I) \vee G$ is regarded as having the form $G \vee (H \wedge I)$.*

(Of the two distribution laws, only the second is needed.) For example, the formula

$$(a \wedge \neg b) \vee \neg (a \vee \neg b)$$

may be converted to CNF by first applying De Morgan's law to the second disjunct,

$$(a \wedge \neg b) \vee (\neg a \wedge b)$$

and then applying distribution,

$$(a \vee \neg a) \wedge (a \vee b) \wedge (\neg b \vee \neg a) \wedge (\neg b \vee b)$$

The two tautologous clauses can be dropped. The precise conversion algorithm appears in Fig. 1.

Exponential growth for this type of conversion can be seen in propositions of the form

$$(a_1 \wedge b_1) \vee \ldots \vee (a_n \wedge b_n) \tag{4}$$

The formula translates to the conjunction of 2^n clauses of the form $L_1 \vee \ldots \vee L_n$, where each L_j is a_j or b_j.

By adding new variables, however, conversion to CNF can be accomplished in linear time. The idea is credited to Tseitin [31] but Wilson's

more compact version [35] simply replaces a disjunction $F \vee G$ with the conjunction

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee F) \wedge (\neg x_2 \vee G),$$

where x_1, x_2 are new variables and the clauses $\neg x_1 \vee F$ and $\neg x_2 \vee G$ encode implications $x_1 \rightarrow F$ and $x_2 \rightarrow G$, respectively. For example, formula (4) yields the conjunction,

$$(x_1 \vee \ldots \vee x_n) \wedge \bigwedge_{j=1}^{n} (\neg x_j \vee a_j) \wedge (\neg x_j \vee b_j),$$

which is equivalent to (4) and grows linearly with the length of (4). The precise algorithm appears in Figure 2.

3.3   Recognizing Redundant Partial Assignments

Let a prime H-implication of a constraint set S be a *prime clausal implication* when H is the set of logical clauses. The following lemma allows one to check for redundancy by examining individual prime implications only.

Lemma 2 *If H contains all logical clauses, then any partial assignment that is redundant for a constraint set S is redundant for some prime H-implication of S. It also violates some prime clausal implication of S.*

   This is easy to see. If a partial assignment $(a_{j_1}, \ldots, a_{j_p}) = (v_1, \ldots, v_p)$ is redundant for S, then S implies the clause

$$a_{j_1}^{1-v_1} \vee \cdots \vee a_{j_p}^{1-v_p} \tag{5}$$

where a_j^0 is the literal $\neg a_j$ and a_j^1 is the literal a_j. So by Lemma 1, some prime implication P of S implies (5). This means the partial assignment is redundant for P. Since (5) is a clause, some prime clausal implication C of S implies (5). Thus the partial assignment violates C.

   By Lemma 2 and the definition of consistency,

Corollary 1 *A constraint set S that contains all of its prime clausal implications is consistent.*

```
Let F be the formula to be converted.
Set k = 0 and S = ∅.
Replace all subformulas of the form G ≡ H with (G → H) ∧ (H → G).
Replace all subformulas of the form G ⊃ H with ¬G ∨ H.
Perform Convert(F).
The CNF form is the conjunction of clauses in S.

Function Convert(F)
   If F has the form C then add F to S.
   Else if F has the form ¬¬G then perform Convert(G).
   Else if F has the form G ∧ H then
      Perform Convert(G) and Convert(H).
   Else if F has the form ¬(G ∧ H) then perform Convert(¬G ∨ ¬H).
   Else if F has the form ¬(G ∨ H) then perform Convert(¬G ∧ ¬H).
   Else if F has the form C ∨ (G ∧ H) then
      Perform Convert(C ∨ G) and Convert(C ∨ H).
   Else if F has the form C ∨ ¬(G ∨ H) then
      Perform Convert(C ∨ (¬G ∧ ¬H)).
   Else if F has the form C ∨ G ∨ H then
      Add C ∨ x_{k-1} ∨ x_{k-2} to S.
      Perform Convert(¬x_{k-1} ∨ G) and Convert(¬x_{k-2} ∨ H).
      Set k = k - 2.
   Else
      Write F as G ∨ H.
      Add x_{k-1} ∨ x_{k-2} to S.
      Perform Convert(¬x_{k-1} ∨ G) and Convert(¬x_{k-2} ∨ H).
```

Figure 2: *Linear-time conversion to CNF (adapted from [20]). The letter C represents any clause. It is assumed that F does not contain variables $x_1, x_2, \ldots$.*

3.4   Consistent Formulations

Given a set S of clauses, the resolution algorithm derives all prime clausal implications of S. So by Corollary 1, resolution achieves consistency.

To apply resolution, the propositions in S must first be converted to clausal form. If each of the two clauses $C, D \in S$ contains exactly one atomic proposition a that appears in the other clause with opposite sign, then the *resolvent* of C and D is the disjunction of all literals that occur in C or D except a and $\neg a$. For instance, the third clause below is the resolvent

```
While S contains clauses C, D with a resolvent R absorbed by
no clause in S:
    Remove from S all clauses absorbed by R.
    Add R to S.
```

Figure 3: *The resolution algorithm applied to clause set S.*

of the first two.

$$\frac{\begin{array}{l} a \vee b \vee c \\ \neg a \vee b \vee \neg d \end{array}}{b \vee c \vee \neg d} \tag{6}$$

A clause C *absorbs* clause D if all the literals of C appear in D. It is clear that C implies D if and only if C absorbs D. The *resolution algorithm* is applied to a set S of clauses by generating all resolvents that are not absorbed by some clause in S, and adding these resolvents to S. It repeats the process until no such resolvents can be generated. The precise algorithm appears in Fig. 3.

Theorem 1 (Quine [28, 29]) *The clause set S' that results from applying the resolution algorithm to clause set S contains precisely the prime clausal implications of S. In particular, S is infeasible if and only if S' consists of the empty clause.*

Resolution applied to the example problem (3) yields the four prime implications

$$\begin{array}{l} \neg a \vee c \\ \neg b \\ \neg d \\ \neg e \end{array} \tag{7}$$

Charles will go to the party if Alice does, and everyone else will stay home. The party will be attended by Charles and Alice, by Charles alone, or by nobody.

The resolution algorithm has exponential worst-case complexity [16] and tends to blow up in practice. Yet it can be very helpful when applied to small constraint sets.

Due to Theorem 1 and Corollary 1,

Corollary 2 *The resolution algorithm achieves consistency.*

14

For instance, (7) can serve as a consistent representation of (3). A number of additional theorems that relate resolution to various types of consistency can be found in [20].

Resolution also achieves hyperarc consistency. The domain of an atomic proposition a can be reduced to $\{1\}$ if and only if the singleton clause a is a prime implication, and it can be reduced to $\{0\}$ if and only if $\neg a$ is a prime implication. In the example (3), the domains of b, d and e are reduced to $\{0\}$. Unfortunately, the entire resolution algorithm must be carried out to achieve hyperarc consistency, so that one may as well aim for full consistency as well.

3.5   Tight MILP Formulations

The feasible set of a proposition can be regarded as 0-1 points in $\mathcal{R}^n$ as well as points in logical space. This means one can design an MILP representation by defining a polyhedron that contains all and only the 0-1 points that satisfy the proposition. The best MILP representation contains inequalities that define the facets of the convex hull of these 0-1 points, as well as equations that define their affine hull.

For example, the proposition $a \equiv b$ has feasible points $(0,0)$ and $(1,1)$. Their affine hull is defined by the equation $a = b$. The convex hull is the line segment from $(0,0)$ to $(1,1)$ and has two facets defined, for example, by $a \geq 0$ and $a \leq 1$. So we have the convex hull MILP representation

$$a = b$$
$$a, b \in \{0, 1\}$$

The inequalities $a \geq 0$ and $a \leq 1$ can be omitted because the linear relaxation of the MILP formulation replaces $a, b \in \{0, 1\}$ with $a, b \in [0, 1]$.

To take another example, the clause $a \vee b$ has feasible points $(0, 1)$, $(1, 0)$ and $(1, 1)$. The facets of the convex hull of these points are $a + b \geq 1$, $a \leq 1$ and $b \leq 1$. The last two inequalities are again redundant, and we have the convex hull MILP representation

$$a + b \geq 1$$
$$a, b \in \{0, 1\}$$

The inequality $a + b \geq 1$ is the only nonelementary facet of the convex hull (i.e., the only facet other than a 0-1 bound on a variable).

One can also derive the inequality $a + b \geq 1$ by observing that since at least one of a, b must be true, the sum of their values must be at least 1. In fact any clause

$$\bigvee_{j \in P} a_j \vee \bigvee_{j \in N} \neg b_j$$

can be given the convex hull formulation

$$\sum_{j \in P} a_j - \sum_{j \in N} (1 - b_j) \geq 1$$

One way to write an MILP formulation of an arbitrary proposition is to convert it to clausal form and write the corresponding inequality for each clause. This might be called the *clausal representation*. Unfortunately, it is not in general a convex hull formulation, even if the clause set is consistent. For instance, the third (and redundant) clause in (6) is not facet-defining even though the clause set is consistent. Nonredundant clauses in a consistent set can also fail to be facet defining, as in the following example:

$$
\begin{array}{ll}
a \vee b & a + b \geq 1 \\
a \vee c & a + c \geq 1 \\
b \vee c & b + c \geq 1
\end{array}
\tag{8}
$$

None of the inequalities defines a facet, and in fact the inequality $a + b + c \geq 2$ is the only nonelementary facet.

Although converting a proposition to clausal form need not give rise to a convex hull formulation, there are some steps one can take to tighten the MILP formulation.

- Convert the proposition to clausal form without adding variables. That is, use the algorithm in Fig. 1 rather than in Fig. 2. As Jeroslow pointed out [23], the addition of new variables can result in a weaker relaxation. For example, $(a \wedge b) \vee (a \wedge c)$ can be converted to the clause set on the left below (using the algorithm of Fig. 1), or to the set on the right (using the algorithm of Fig. 2):

$$
\begin{array}{cc}
& x_1 \vee x_2 \\
& \neg x_1 \vee a \\
a & \neg x_1 \vee b \\
b \vee c & \neg x_2 \vee a \\
& \neg x_2 \vee c
\end{array}
$$

  The relaxation of the first set fixes $a = 1$, but the relaxation of the second set only requires that $a \in [1/2, 1]$.

16

- Apply the resolution algorithm to a clause set before converting the clauses to inequalities. It is easily shown that resolvents correspond to valid inequalities that tighten the relaxation. The clausal representation of problem (3), for example, is much tighter (in fact, it is a convex hull formulation) if one first applies resolution to generate the consistent formulation (7):

$$a \leq c$$
$$b = d = e = 0$$

Resolution can even tighten the relaxation to the point that it is infeasible. Consider for example the clause set on the left below, whose relaxation is shown on the right.

$$
\begin{array}{ll}
a \vee b & a + b \geq 1 \\
\neg a \vee b & (1 - a) + b \geq 1 \\
a \vee \neg b & a + (1 - b) \geq 1 \\
\neg a \vee \neg b & (1 - a) + (1 - b) \geq 1 \\
& a, b \in [0, 1]
\end{array}
$$

The relaxation has the feasible solution $(a, b) = (1/2, 1/2)$. Yet resolution reduces the clause set to the empty clause, whose relaxation $0 \geq 1$ is infeasible. This shows that resolution has greater power for detecting infeasibility than linear relaxation.

- If it is computationally difficult to apply the full resolution algorithm, use a partial algorithm. For instance, one might generate all resolvents have length less than k for some small k. One might also apply *unit resolution*, which is the resolution algorithm modified so that at least one clause of every pair of clauses resolved is a unit clause (i.e., consists of one literal). It can be shown that unit resolution generates precisely the clauses that correspond to rank 1 Chvátal cutting planes [13].

- To obtain further cutting planes, view the clauses as a special case of cardinality clauses, and apply the inference algorithm of Section 4 below. This procedure derives the facet-defining inequality $a + b + c \geq 2$ for (8), for instance.

Table 1 displays the prime implications and a convex hull formulation of some simple propositions. For propositions 1–2 and 4–8, the prime implications correspond to a convex hull formulation. Formula 7 is very common in

17

MILP modeling, because it asks one to linearize the relation $ab = c$, which occurs when condition c is true if and only both a and b are true. One might be tempted to write the MILP model

$$c \geq a + b - 1$$
$$2c \leq a + b$$

The first inequality is facet defining, but the second is not.

The facet-defining inequality for proposition 9 is the smallest example (smallest in terms of the dimension of the space) of a nonclausal facet-defining inequality for set of 0-1 points. The inequality of proposition 10 is the smallest example of a facet-defining inequality with a coefficient not in $\{0, 1, -1\}$.

4 Cardinality Clauses

A cardinality clause states that at least k of a set of literals is true. Cardinality clauses retain some of the properties of propositional logic while making easier to model problems that require counting.

4.1 Basic Properties

A cardinality clause can be written

$$\{L_1, \ldots, L_m\} \geq k \tag{9}$$

for $m \geq 0$ and $k \geq 1$, where the L_js are literals containing distinct variables. The clause asserts that at least k of the literals are true. To assert that at most k literals are true, one can write

$$\{\neg L_1, \ldots, \neg L_m\} \geq m - k$$

By convention a cardinality clause (9) with $m < k$ is necessarily false. (9) is a *classical clause* if $k = 1$.

The propositional representation of (9) requires $\binom{m}{k-1}$ clauses:

$$\bigvee_{j \in J} L_j, \quad \text{all } J \subset \{1, \ldots, m\} \text{ with } |J| = m - k + 1 \tag{10}$$

It is useful to write cardinality clause (9) in the form $A \geq k$, where A is a set of literals.

18

Table 1: Prime implications and convex hull formulations of some simple propositions. The set of prime implications of a proposition can serve as a consistent formulation of that proposition.

	Proposition	Prime Implications	Convex Hull Formulation
1.	$a \vee b$	$a \vee b$	$a + b \geq 1$
2.	$a \to b$	$\neg a \vee b$	$a \leq b$
3.	$a \equiv b$	$\neg a \vee b$	$a = b$
		$a \vee \neg b$	
4.	$a \to (b \vee c)$	$\neg a \vee b \vee c$	$a \leq b + c$
5.	$(a \vee b) \to c$	$\neg a \vee b$	$a \leq b$
		$\neg a \vee c$	$a \leq c$
6.	$(a \vee b) \equiv c$	$a \vee b \vee \neg c$	$c \leq a + b$
		$\neg a \vee c$	$a \leq c$
		$\neg b \vee c$	$b \leq c$
7.	$(a \wedge b) \equiv c$	$\neg a \vee \neg b \vee c$	$c \geq a + b - 1$
		$\neg a \vee c$	$a \geq c$
		$\neg b \vee c$	$b \geq c$
8.	$(a \equiv b) \equiv c$	$a \vee b \vee c$	$a + b + c \geq 1$
		$a \vee \neg b \vee \neg c$	$b + c \leq a - 1$
		$\neg a \vee b \vee \neg c$	$a + c \leq b - 1$
		$\neg a \vee \neg b \vee c$	$a + b \leq c - 1$
9.	$(a \vee b) \wedge$	$a \vee b$	$a + b + c \geq 2$
	$(a \vee c) \wedge$	$a \vee c$	
	$(b \vee c)$	$b \vee c$	
10.	$(a \vee b \vee c) \wedge$	$a \vee b \vee c$	$a + b + c + 2d \geq 3$
	$(a \vee d) \wedge$	$a \vee d$	
	$(b \vee d) \wedge$	$b \vee d$	
	$(c \vee d)$	$c \vee d$	

Lemma 3 ([12]) *$A \geq k$ implies $B \geq \ell$ if and only if $|A \setminus B| \leq k - \ell$.*

For example, $\{a, \neg b, c, d, e\} \geq 4$ implies $\{a, \neg b, c, \neg d\} \geq 2$.

A modeling example might go as follows. A firm wishes to hire one or more employees from a pool of four applicants (Alice, Betty, Charles, Diane), including one manager. Only one of the applicants (Betty) does not belong to a minority group. The firm must observe the following constraints.

$$
\begin{array}{lll}
\text{At least two women must be hired.} & \{a, b, d\} \geq 2 & (i) \\
\text{At least two minority applicants} & \{a, c, d\} \geq 2 & (ii) \\
\quad \text{must be hired.} & & \\
\text{Only Alice and Betty are qualified} & \{a, b\} \geq 1 & (iii) \qquad (11) \\
\quad \text{to be managers.} & & \\
\text{Alice won't work for the firm unless} & \{\neg a, c\} \geq 1 & (iv) \\
\quad \text{it hires Charles.} & &
\end{array}
$$

4.2   Consistency and Prime Implications

Consistency can be achieved for a single cardinality clause simply by generating all of the classical clauses it implies, using (10). However, there is no need to do so, since it is easy to check whether a partial assignment is redundant for a given cardinality clause (9): it is redundant if it falsifies more than $m - k$ literals in $\{L_1, \ldots, L_m\}$.

One can achieve consistency for a set S of cardinality clauses by generating all the classical clauses S implies and applying the standard resolution algorithm to these clauses. However, this could result in a large number of clauses.

A more efficient way to recognize redundant partial assignments is to generate *prime cardinality implications* of S; that is, prime H-implications where H is the set of cardinality clauses. Since H contains clauses, Lemma 2 guarantees that any redundant partial assignment for S is redundant for a prime cardinality clause of S.

Prime cardinality implications can be generated with a procedure one might call *cardinality resolution* [18], which is a specialization of the 0-1 resolution procedure described in the next section. Cardinality resolution generates two kinds of resolvents: classical resolvents and diagonal sums.

A cardinality clause set S has a *classical resolvent R* if

(a) there are classical clauses C_1, C_2 with resolvent R such that each C_i is implied by a cardinality clause in S, and

(b) no cardinality clause in S implies R.

Thus either clause below the line is a classical resolvent of the set S of clauses above the line:

$$
\frac{
\begin{array}{c}
\{a, b, c\} \geq 2 \\
\{\neg a, c, d\} \geq 2
\end{array}
}{
\begin{array}{c}
\{b, d\} \geq 1 \\
\{c\} \geq 1
\end{array}
} \tag{12}
$$

The clause $\{b, d\} \geq 1$, for example, is the resolvent of $\{a, b\} \geq 1$ and $\{\neg a, d\} \geq 1$, each of which is implied by a clause in S.

The cardinality clause set S has a *diagonal sum* $\{L_1, \ldots, L_m\} \geq k + 1$ if

(a) there are cardinality clauses $A_j \geq k$ for $j = 1, \ldots, m$ such that each $A_j \geq k$ is implied by some $C_j \in S$,

(b) each $A_j \subset \{L_1, \ldots, L_m\}$,

(c) $L_j \notin A_j$ for each j, and

(d) no cardinality clause in S implies $\{L_1, \ldots, L_m\} \geq k + 1$.

Consider for example the following set S of cardinality clauses:

$$
\begin{array}{c}
\{a, b, e\} \geq 2 \\
\{b, c, e\} \geq 2 \\
\{a, c, d\} \geq 1
\end{array} \tag{13}
$$

A diagonal sum $\{a, b, c, d\} \geq 2$ of S can be derived from the following clauses $A_j \geq k$:

$$
\begin{array}{c}
\{\quad b, c, d\,\} \geq 1 \\
\{a, \quad c, d\,\} \geq 1 \\
\{a, b, \quad d\,\} \geq 1 \\
\{a, b, c, \quad\} \geq 1
\end{array} \tag{14}
$$

Note that each clause in (14) is implied by (at least) one of the clauses in S.

The cardinality resolution algorithm appears in Fig. 4. The following theorem is a special case of the 0-1 resolution theorem stated in the next section.

Theorem 2 ([18, 19]) *The set that results from applying the cardinality resolution algorithm to cardinality clause set S contains precisely the prime cardinality implications of S. Consistency can be achieved for S by applying cardinality resolution algorithm but generating only classical resolvents in the course of the algorithm.*

```
While S has a classical resolvent or diagonal sum R
that is implied by no cardinality clause in S:
   Remove from S all cardinality clauses implied by R.
   Add R to S.
```

Figure 4: *The cardinality resolution algorithm applied to cardinality clause set S.*

The theorem can be illustrated with example (11). Clauses (ii) and (iv) yield the classical resolvent

$$\{c\} \geq 1 \tag{15}$$

Each of the following diagonal sums can be derived from (i) and (15):

$$\begin{aligned} \{a, b, c\} &\geq 2 \\ \{b, c, d\} &\geq 2 \end{aligned} \tag{16}$$

Now the clauses (i),(ii) and the two clauses in (16) yield the diagonal sum

$$\{a, b, c, d\} \geq 3 \tag{17}$$

The prime implications of (11) consist of the two clauses (15) and (17). They say that the firm must hire at least three people, and it must hire Charles. These prime implications also allow one to recognize all redundant partial assignments for (11). For instance, the partial assignment $c = 0$ is redundant for (and in fact violates) the prime implication (15) but is not redundant for any of the original constraints (11).

Since diagonal summation is not needed to achieve consistency, it does not allow one to recognize any additional redundant partial assignments. For instance, all partial assignments that are redundant for $\{a, b, c, d\} \geq 2$ are already redundant for the clauses in (13). Yet generation of diagonal sums can accelerate the process of achieving consistency, and it can reduce the number of constraints one much check to determine whether a partial assignment is redundant.

A possibly faster way to recognize *some* redundant partial assignments is to generate a set 0-1 inequalities that are known to imply resolvents. A cardinality clause

$$\{a_1, \ldots, a_m, \neg b_1, \ldots, \neg b_n\} \geq k \tag{18}$$

is easily written as a 0-1 inequality:

$$\sum_{j=1}^{m} a_j - \sum_{j=1}^{n} (1 - b_j) \geq k \tag{19}$$

22

It is convenient to say that a 0-1 inequality implies (18) if it implies (19).

Theorem 3 (Barth [8]) *All classical resolvents of two cardinality clauses are implied by the sum of the two corresponding 0-1 inequalities.*

An alternative proof of the theorem appears in [20]. As an example, the sum of the inequalities that correspond to the cardinality clauses in (12) appears below the line:

$$a + b + c \geq 2$$
$$\underline{(1 - a) + c + d \geq 2} \tag{20}$$
$$b + 2c + d \geq 3$$

Both classical resolvents in (12) are implied by this sum. The sum (20) also helps one to recognize redundant partial assignments. For example, $c = 0$ is redundant for $b + 2c + d \geq 3$ but not for either of the clauses in (12).

It is easy to check whether a partial assignment

$$(x_{j_1}, \ldots, x_{j_p}) = (v_1, \ldots, v_p)$$

is redundant for a 0-1 inequality $\alpha x \geq \beta$. It is redundant if and only if $\alpha \bar{x} < \beta$, where

$$\bar{x}_j = \begin{cases} v_j & \text{if } j \in \{j_1, \ldots, j_p\} \\ 1 & \text{if } j \notin \{j_1, \ldots, j_p\} \text{ and } \alpha_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

Theorem 3 is paralleled by a similar result for diagonal sums, but it does not assist in the recognition of redundant partial assignments.

Theorem 4 (Barth [8]) *Any diagonal sum $A \geq k + 1$ of a cardinality clause set S is implied by the sum of $|A|$ or fewer cardinality clauses in S.*

Again an alternate proof appears in [20]. For example, the diagonal sum $\{a, b, c, d\} \geq 2$ derived from the set S in (13) is implied by the sum of the three clauses in S, which is

$$2a + 2b + 2c + d + 2e \geq 5$$

Theorems 3 and 4 do not in general produce prime implications, because the sums they describe imply only cardinality clauses that appear in the first round of resolution.

4.3 Tight MILP Relaxations

The single 0-1 inequality (19) is a convex hull formulation of the cardinality clause (18) [37]. Tighter relaxations can be obtained by applying the cardinality resolution algorithm, in whole or in part, and writing the resolvents as 0-1 inequalities. Thus the example (11) has the MILP formulation

$$a + b + c + d \geq 3$$
$$c = 1$$
$$a, b, c \geq \{0, 1\}$$

This happens to be a convex hull formulation.

Generating sums as indicated in Theorems 3 does not strengthen the MILP formulation, even though it strengthens the logical formulation by excluding more redundant partial assignments.

5   0-1 Inequalities

One can regard 0-1 inequalities as logical propositions as well as numerical constraints. A 0-1 inequality is a linear inequality of the form $ax \geq \alpha$ where each $x_j \in \{0, 1\}$. We will suppose that α and each coefficient a_j are integers.

There is a large literature (surveyed in [26, 36]) that investigates how to tighten a 0-1 formulation by adding cutting planes. It is also possible to write a good 0-1 formulation for logical purposes. In particular there is a resolution procedure that yields all prime 0-1 implications of a set of 0-1 inequalities.

The 0-1 resolution procedure requires that one be able to check whether one 0-1 inequality implies another, but unfortunately this is an NP-complete problem. Checking whether $ax \geq \alpha$ implies $bx \geq \beta$ is equivalent to checking whether the minimum of bx subject to $ax \geq \alpha, x \in \{0, 1\}^n$ is at least β. The latter is the general 0-1 knapsack problem, which is NP-complete. Due to this difficulty, 0-1 resolution is most likely to be useful when applied to a special class of 0-1 inequalities, such as cardinality clauses.

Let a *clausal inequality* be a 0-1 inequality that represents a logical clause. Two clausal inequalities resolve in a manner analogous to the corresponding clauses. A set S of 0-1 inequalities has a *classical resolvent R* if R is a clausal inequality and

(a) there are clausal inequalities C_1, C_2 with resolvent R such that each C_i is implied by an inequality in S, and

(b) no inequality in S implies R.

A 0-1 inequality $ax \geq \alpha$ implies a clause $x_{j_1}^{v_1} \vee \cdots \vee x_{j_k}^{v_k}$ if and only if the partial assignment $(x_{j_1}, \ldots, x_{j_k}) = (1 - v_1, \ldots, 1 - v_k)$ is redundant for $ax \geq \alpha$, which can be checked as described earlier.

To take an example, either clausal inequality below the line is a classical resolvent of the clauses above the line:

$$x_1 + 2x_2 + x_3 \geq 2$$
$$\underline{x_1 - 2x_2 + x_3 \geq 0}$$
$$x_1 \geq 1$$
$$x_3 \geq 1$$

To define diagonal sums, it is convenient to write a 0-1 inequality $ax \geq \alpha$ in the form $ax \geq \delta - n(a)$, where $n(a)$ is the sum of the negative components of a and δ is the *degree* of the inequality. Thus clausal inequalities have degree 1. A set S of 0-1 inequalities has a *diagonal sum* $ax \geq \delta + 1 + n(a)$ if there is a nonempty $J \subset \{1, \ldots, n\}$ such that

(a) $a_j = 0$ for $j \notin J$,

(b) there are 0-1 inequalities $a^j x \geq \delta - n(a^j)$ for $j \in J$ such that each $a^j x \geq \delta - n(a^j)$ is implied by some inequality in S,

(c) each $a^i x \geq \beta + n(a^i)$ satisfies

$$a_j^i = \begin{cases} a_j - 1 & \text{if } j = i \text{ and } a_j > 0 \\ a_j + 1 & \text{if } j = i \text{ and } a_j < 0 \\ a_j & \text{otherwise} \end{cases}$$

(d) no inequality in S implies $ax \geq \delta + 1 + n(a)$.

Consider for example the set S consisting of

$$\begin{aligned} x_1 + 5x_2 - 3x_3 + x_4 &\geq 4 - 3 \\ 2x_1 + 4x_2 - 3x_3 + x_4 &\geq 4 - 3 \\ 2x_1 + 5x_2 - 2x_3 + x_4 &\geq 4 - 2 \\ 2x_1 + 5x_2 - 3x_3 \quad\;\; &\geq 4 - 3 \end{aligned} \qquad (21)$$

Note that the inequalities are identical except that the diagonal element in each is reduced by one in absolute value. Since each inequality in (21) is implied by a member of S (namely, itself), S has the diagonal sum

$$2x_1 + 5x_2 - 3x_3 + x_4 \geq 5 - 3$$

25

```
While S has a classical resolvent or diagonal sum R
that is implied by no inequality in S:
   Remove from S all inequalities implied by R.
   Add R to S.
```

Figure 5: *The 0-1 resolution algorithm applied to set S of 0-1 inequalities.*

in which the degree is increased by one.

The 0-1 resolution algorithm appears in Fig. 5. The completeness theorem for 0-1 resolution must be stated carefully, since there are infinitely many 0-1 inequalities in a given set of variables $x = (x_1, \ldots, x_n)$. Let a finite set H be a *monotone* set of 0-1 inequalities if H contains all logical clauses and given any inequality $ax \geq \delta + n(a)$ in H, H contains all 0-1 inequalities $a'x \geq \delta' + n(a')$ such that $|a'_j| \leq |a_j|$ for all j and $0 \leq \delta' \leq \delta$.

Theorem 5 ([19]) *If H is a monotone set of 0-1 inequalities, the set that results from applying the 0-1 resolution algorithm to a set $S \in H$ of 0-1 inequalities contains precisely the prime H-implications of S.*

Again, diagonal summation does not allow one to recognize additional redundant assignments, although it can reduce the number of inequalities one must examine to check for redundancy.

6 Cardinality Rules

A cardinality rule is a generalized form of cardinality clause that provides more expressive power. A cardinality rule has the form

$$\{L_1, \ldots, L_m\} \geq k \rightarrow \{M_1, \ldots, M_n\} \geq \ell \qquad (22)$$

where literals L_i and M_j contain distinct variables. The rule states that if at least k of the literals L_i are true, then at least ℓ of the literals M_j are true. It is assumed that $m \geq k \geq 0$, $n \geq 0$, and $\ell \geq 1$.

No specialized inference method has been developed for cardinality rules, but a convex hull MILP formulation for an individual cardinality rule is known. Without loss of generality, assume all literals in (22) are positive:

$$\{a_1, \ldots, a_m\} \geq k \rightarrow \{b_1, \ldots, b_n\} \geq \ell \qquad (23)$$

Any negated literal $\neg a_i$ is represented in the convex hull formulation by $1 - a_i$ rather than a_i.

26

```
Procedure Facet({a_1,...,a_m} ≥ k → {b_1,...,b_n} ≥ ℓ)
If m > k = 1 then
    For all i ∈ {1,...,m} perform Facet(a_i → {b_1,...,b_n} ≥ ℓ).
Else if n = ℓ > 1 then
    For all j ∈ {1,...,n} perform Facet({a_1,...,a_m} ≥ k → b_j).
Else
    Generate the facet-defining inequality
        ℓ(a_1 + ··· + a_m) − (m − k + 1)(b_1 + ... + b_n) ≤ ℓ(k − 1).
    If m > k and n > 0 then
        For all {i_1,...,i_{m−1}} ⊂ {1,...,m},
            perform Facet({a_{i_1},...,a_{i_{m−1}}} ≥ k → {b_1,...,b_n} ≥ ℓ).
    If ℓ > 1 and m > 0 then
        For all {j_1,...,j_{n−1}} ⊂ {1,...,n},
            Perform Facet({a_1,...,m} ≥ k → {b_{j_1},...,b_{j_{n−1}}} ≥ ℓ − 1).
```

Figure 6: *An algorithm, adapted from [37], for generating a convex hull formulation of the cardinality rule (23). It is assumed that $a_i, b_j \in \{0,1\}$ is part of the formulation. The cardinality clause $\{a_i\} \geq 1$ is abbreviated a_i. The procedure is activated by calling it with (23) as the argument.*

Theorem 6 (Yan, Hooker [37]) *The algorithm of Fig. 6 generates a convex hull formulation for the cardinality rule (23).*

This result has been generalized in [4].

The following example is presented in [37]. Let a_i state that a plant is built at site i for $i = 1, 2$, and let b_j state that product j is made for $j = 1, 2, 3$. Plant construction must observe the following constraints.

If at least 2 plants are built, $\quad \{a_1, a_2, a_3\} \geq 2 \rightarrow \{b_1, b_2, b_3\} \geq 2$
at least 2 new products
should be made.

At most 1 new product should $\quad \{\neg a_1, \neg a_2\} \geq 1 \rightarrow \{\neg b_1, \neg b_2, \neg b_3\} \geq 2$
be made, unless plants
are built at both sites
1 and 2.

$$(24)$$

The convex hull formulation for the first rule is

$$
\begin{aligned}
2(a_1 + a_2 + a_3) - 2(b_1 + b_2 + b_3) &\geq 2 \\
7(a_1 + a_2) - b_1 - b_2 - b_3 &\geq 2 \\
2(a_1 + a_3) - b_1 - b_2 - b_3 &\geq 2 \\
2(a_2 + a_3) - b_1 - b_2 - b_3 &\geq 2 \\
a_1 + a_2 + a_3 - 2(b_1 + b_2) &\geq 1 \\
a_1 + a_2 + a_3 - 2(b_1 + b_3) &\geq 1 \\
a_1 + a_2 + a_3 - 2(b_2 + b_3) &\geq 1 \\
a_1 + a_2 - b_1 - b_2 &\geq 1 \\
a_1 + a_2 - b_1 - b_3 &\geq 1 \\
a_1 + a_2 - b_2 - b_3 &\geq 1 \\
a_1 + a_3 - b_1 - b_2 &\geq 1 \\
a_1 + a_3 - b_1 - b_3 &\geq 1 \\
a_1 + a_3 - b_2 - b_3 &\geq 1 \\
a_2 + a_3 - b_1 - b_2 &\geq 1 \\
a_2 + a_3 - b_1 - b_3 &\geq 1 \\
a_2 + a_3 - b_2 - b_3 &\geq 1 \\
a_i, b_j &\in \{0,1\}
\end{aligned}
$$

The convex hull formulation for the second rule in (24) is

$$
\begin{aligned}
2(1 - a_1) - (1 - b_1) - (1 - b_2) - (1 - b_3) &\geq 0 \\
2(1 - a_2) - (1 - b_1) - (1 - b_2) - (1 - b_3) &\geq 0 \\
(1 - a_1) - (1 - b_1) - (1 - b_2) &\geq 0 \\
(1 - a_1) - (1 - b_1) - (1 - b_3) &\geq 0 \\
(1 - a_1) - (1 - b_2) - (1 - b_3) &\geq 0 \\
(1 - a_2) - (1 - b_1) - (1 - b_2) &\geq 0 \\
(1 - a_2) - (1 - b_1) - (1 - b_3) &\geq 0 \\
(1 - a_2) - (1 - b_2) - (1 - b_3) &\geq 0 \\
a_i, b_j &\in \{0,1\}
\end{aligned}
$$

7 Mixing Logical and Continuous Variables

Very often logical and continuous variables must be combined in a constraint. A constraint involving continuous variables, for example, may be enforced only when a certain logical expression is true. One versatile syntax for expressing such constraints is a *conditional constraint*:

$$
g(a) \rightarrow Ax \geq b \tag{25}
$$

where $g(a)$ is a formula of propositional logic containing atomic propositions $a = (a_1, \ldots, a_n)$. Formula (25) says that $Ax \geq a$ is enforced if $g(a)$ is true.

It is straightforward for a solver to implement conditional constraints. The constraint in the consequent of (25) is posted whenever the antecedent $g(a)$ becomes true in the course of branching and constraint propagation. The rule is "fired" when the currently fixed atomic propositions a_j are enough to ensure that $g(a)$ is true; that is, when every extension of the current partial assignment makes $g(a)$ true. It is clearly advantageous to put the antecedent in clausal form, since this allows one to check whether every extension makes $g(a)$ true by checking whether the partial assignment already satisfies every clause. For example, the partial assigment $a_1 = 1$ necessarily satisfies the clause set $\{a_1 \vee a_2, a_1 \vee a_3\}$ because it makes a literal in both clauses true, whereas $a_2 = 1$ does not necessarily satisfy the clause set.

Relaxations can also be generated for conditional constraints, as will be seen below.

7.1   Modeling with Conditional Constraints

To illustrate the use of conditional constraints, consider a fixed charge problem in which an activity incurs no cost when its level $x = 0$ and a fixed cost F plus variable cost cx when $x > 0$. This is normally formulated in MILP with the help of a big-M constraint:

$$z = fa + cx$$
$$x \leq Ma$$
$$x \geq 0, a \in \{0, 1\}$$

where M is the maximum level of the activity and z is the total cost. Binary variable a is 1 when the activity level is positive and zero otherwise. A conditional formulation could be written

$$a \rightarrow z \geq f + cx$$
$$\neg a \rightarrow x = 0 \qquad (26)$$
$$x, z \geq 0$$

To take another situation, suppose it costs f_1 to build plant A only, f_2 to build plant B only, and f_3 to build both plants, where $f_3 \neq f_1 + f_2$. The capacities of plants A and B are M_a and M_b, respectively, and total production is x. One might write the total cost z in an MILP by defining a 0-1 variables a that takes the value 1 when plant A is built, and similarly

for b. A third 0-1 variable c is 1 when $a = b = 1$. Thus $c = ab$, an expression that must be linearized. The model becomes

$$z = af_1 + bf_2 + c(f_3 - f_1 - f_2)$$
$$c \geq a + b - 1$$
$$a \geq c$$
$$b \geq c$$
$$x \leq aM_a + bM_b$$
$$a, b, c \in \{0, 1\}$$

With a little thought one might realize that the cost constraint simplifies if a is set to 1 when *only* plant A is built, and similarly for $b = 1$, so that $a + b \leq 1$. Also $a = b = 0$ when $c = 1$, which can be written with the constraints $a + c \leq 1$ and $b + c \leq 1$. These three inequalities can be combined into a single inequality $a + b + c \leq 1$. This simplifies the model but sligtly complicates the capacity constraint.

$$z = af_1 + bf_2 + cf_3$$
$$a + b + c \leq 1$$
$$x \leq aM_a + bM_b + c(M_a + M_b) \tag{27}$$
$$a, b, c \in \{0, 1\}$$

A logic-based approach permits one to write the model as it is originally conceived:

$$(\neg a \wedge \neg b) \rightarrow (x = z = 0)$$
$$(a \wedge \neg b) \rightarrow (x \leq M_a, z = f_a)$$
$$(\neg a \wedge b) \rightarrow (x \leq M_b, z = f_b) \tag{28}$$
$$(a \wedge b) \rightarrow (x \leq M_a + M_b, z = f_c)$$

7.2   Relaxations for Conditionals

MILP formulations have the advantage that a continuous relaxation is readily available from the model itself, simply by dropping integrality constraints. However, conditional constraints can trigger the generation of comparable or even better relaxations.

To obtain a relaxation, however, one must indicate which groups of conditionals imply a disjuncion (a chore that could be done automatically if desired). A natural way to indicate this is to use an *inequality-or* global constraint. The two conditionals (26), for example, imply a disjunction: either a is true and $z \geq f + cx$, or a is false and $x = 0$. This can be written

$$\text{inequality-or}\left\{ \begin{pmatrix} a \\ z \geq f + cx' \end{pmatrix}, \begin{pmatrix} \neg a \\ x = 0 \end{pmatrix} \right\} \tag{29}$$

Similarly, the conditionals (28) can be written

$$\text{inequality-or}\left\{\begin{pmatrix}\neg a \wedge \neg b \\ x = z = 0\end{pmatrix}, \begin{pmatrix} a \wedge \neg b \\ x \le M_a \\ z = f_a\end{pmatrix}, \begin{pmatrix}\neg a \wedge b \\ x \le M_b \\ z = f_b\end{pmatrix}, \begin{pmatrix}\neg a \wedge \neg b \\ x \le M_a + M_b \\ z = f_c\end{pmatrix}\right\} \tag{30}$$

In general inequality-or has the form

$$\text{inequality-or}\left\{\begin{pmatrix} g_1(a) \\ A^1 x \ge b^{1,}\end{pmatrix}, \ldots, \begin{pmatrix} g_k(a) \\ A^K x \ge b^K\end{pmatrix}\right\} \tag{31}$$

Constraint (31) implies a disjunction of linear systems

$$\bigvee_{k=1}^{K} A^k x \ge b^k \tag{32}$$

that can be relaxed in two ways. The big-M relaxation is

$$A^k x \ge b^k - M^k(1 - y_k), \quad k = 1, \ldots, K$$
$$\sum_{k=1}^{K} y_k = 1 \tag{33}$$
$$y_k \ge 0, \quad k = 1, \ldots, K$$

where
$$M_i^k = b_i^k - \min_k\left\{\min_x\left\{A_i^k x \mid A_i^{k'} \le b_i^{k'}\right\}, \text{ all } k' \ne k\right\}$$

If some $M_i^k = \infty$, then x must be bounded to prevent this. Beaumont [9] showed that the relaxation (31) simplifies for a disjunction of inequalities

$$\bigvee_{k=1}^{K} a^k x \ge b_k$$

where $0 \le x \le m$. The relaxation becomes a single inequality

$$\left(\sum_{k=1}^{K} \frac{a^k}{M_k}\right) x \ge \sum_{k=1}^{K} \frac{b_k}{M_k} - K + 1 \tag{34}$$

where

$$M_k = b_k - \min_k\left\{\min_x\left\{a_i^k x \mid a_i^{k'} \le b_{k'}, \ 0 \le x \le m\right\}, \text{ all } k' \ne k\right\} \tag{35}$$

A second relaxation for (32) is the disjunctive convex hull relaxation of Balas [2, 3]:

$$A^k x^k \geq b^k y_k, \quad k = 1, \ldots, K$$
$$x = \sum_{k=1}^{K} x^k$$
$$\sum_{k=1}^{K} y_k = 1 \tag{36}$$
$$y_k \geq 0, \quad k = 1, \ldots, K$$

The disjunctive convex hull relaxation provides the tightest linear relaxation but requires additional variables $x^1, \ldots, x^K$.

For example, the disjunction in (29) can be written

$$(-cx + z \geq f) \vee (-x \geq 0) \tag{37}$$

where $0 \leq x \leq M$ (it turns out that no bound on z is needed). To apply Beaumont's big-M relaxation (34) we obtain from (35) that $M_1 = f$ and $M_2 = M$. Thus the big-M relaxation is

$$z \geq \left(c + \frac{f}{M} \right) x \tag{38}$$

The disjunctive convex hull relaxation (36) of (37) simplifies to

$$z \geq fy + cx, \quad 0 \leq y \leq 1 \tag{39}$$

This is equivalent to the big-M relaxation (38), which is the projection of (39) onto x, z. For this particular model, the big-M relaxation is as tight as the disjunctive convex hull relaxation and is simpler.

Sometimes the disjunctive convex hull relaxation is simpler than the big-M relaxation. Consider for example a problem in which one can install either of two machines, or neither. Machine A has cost 50 and capacity 5, while machine B has cost 80 and capacity 10. If z is the total fixed cost and x the output, the model is

$$\text{inequality-or} \left\{ \begin{pmatrix} \neg a \wedge \neg b \\ x = 0 \end{pmatrix}, \begin{pmatrix} a \\ z \geq 50 \\ 0 \leq x \leq 5 \end{pmatrix}, \begin{pmatrix} b \\ z \geq 80 \\ 0 \leq x \leq 10 \end{pmatrix} \right\}$$

The big-M relaxation is

$$x \leq 10(y_1 + y_2)$$
$$x \leq 5(2 - y_1)$$
$$x \leq 5(1 + y_2)$$
$$z \geq 50y_1 \qquad\qquad (40)$$
$$z \geq 80y_2$$
$$y_1 + y_2 \leq 1$$
$$x, y_1, y_2 \geq 0$$

which projects onto x, z as follows:

$$z \geq (40/13)x$$
$$z \geq 16x - 80$$
$$0 \leq x \leq 10$$

The disjunctive convex hull relaxation is simpler than (40):

$$x \leq 5y_1 + 10y_2$$
$$z \leq 50y_1 + 80y_2$$
$$y_1 + y_2 \leq 1$$
$$a, y_1, y_2 \geq 0$$

It is also tighter, since it projects onto x, z as follows:

$$z \geq 8x$$
$$0 \leq x \leq 10$$

Typically, however, the disjunctive convex hull relaxation is more complicated than the big-M relaxation.

Finally, the disjunctive convex hull relaxation of the plant building constraint (30) is precisely the linear relaxation of (27).

8   Additional Global Constraints

The CP community has developed a large lexicon of global constraints. Three particularly useful ones are briefly examined here: *all-different*, *element* and *cumulative*. All have been studied with respect to both relaxations and consistency maintenance. A number of hybrid modeling examples using these and other global constraints may be found in [20].

8.1 The All-different Constraint

The all-different constraint is ubiquitous in scheduling applications. It is written

$$\text{all-different}\{y_1, \ldots, y_k\} \tag{41}$$

where $y_1, \ldots, y_k$ are variables with finite domains. The constraint states that $y_1, \ldots, y_k$ take distinct values.

The all-different constraint permits an elegant formulation of the traveling salesman problem. The object is to find the cheapest way to visit each of n cities exactly once and return home. Let y_i be the ith city in the tour, and let c_{jk} be the cost of traveling from city j to city k. The problem can be written

$$\min \quad \sum_{i=1}^{n} c_{y_i y_{i+1}} \tag{42}$$
$$\text{subject to} \quad \text{all-different}\{y_1, \ldots, y_n\}$$

where y_{n+1} is identified with y_1. (It may be more practical to write the traveling salesman problem with the *cycle* constraint [20], since it can be relaxed using well-known cutting planes for the problem.)

The best-known domain reduction algorithm, that of Régin [30], achieves hyperarc consistency. Let G be a bipartite graph whose vertices correspond to the variables $y_1, \ldots, y_k$ and to the elements $(v_1, \ldots, v_n)$ in the union of the variable domains $D_1, \ldots, D_k$. G contains an edge (v_j, v_k) whenever $v_k \in D_j$. Hyperarc consistency is obtained as follows. First find a maximum cardinality bipartite matching on G. For each vertex that is not covered by the matching, mark all edges that are part of an alternating path that starts at that vertex. By a theorem of Berge [10], these edges belong to at least one, but not every, maximum cardinality matching. For the same reason, mark every edge that belongs to some alternating cycle. (An alternating path or cycle is one in which every second edge belongs to the matching.) Now every unmarked edge in the matching belongs to every maximum cardinality matching, by Berge's theorem. One can therefore delete all unmarked edges from G that are not part of the matching. The reduced domains D_j contains values v_k for which (x_j, v_k) is a remaining edge.

For example, consider the constraint all-different$(y_1, \ldots, y_5)$ in which the

initial domains are as shown on the left below.

$$
\begin{aligned}
D_1 &= \{1\} & &\to \{1\} \\
D_2 &= \{2, 3, 5\} & &\to \{2, 3\} \\
D_3 &= \{1, 2, 3, 5\} & &\to \{2, 3\} \\
D_4 &= \{1, 5\} & &\to \{5\} \\
D_5 &= \{1, 2, 3, 4, 5, 6\} & &\to \{4, 6\}
\end{aligned}
$$

The algorithm reduces the domains to those shown on the right.

The convex hull relaxation for all-different is known for the case when the variables have the same domains, even if it is rather weak and exponentially long [20, 34]. Let each domain D_j be a set $\{t_1, \ldots, t_n\}$ of numerical values, with $t_1 \leq \ldots \leq t_n$. A convex hull relaxation of (41) can be written as follows:

$$
\begin{aligned}
\sum_{j=1}^{n} x_j &= \sum_{j=1}^{n} t_j \\
\sum_{j \in J} x_j &\geq \sum_{j=1}^{|J|} t_j, \quad \text{all } J \in \{1, \ldots, n\} \text{ with } |J| < n
\end{aligned}
\tag{43}
$$

For example, consider all-different$\{y_1, y_2, y_3\}$ with domain $\{15, 20, 25\}$. The convex hull relaxation is

$$
\begin{aligned}
y_1 + y_2 + y_3 &= 60 \\
y_1 + y_2 &\geq 35 \\
y_1 + y_3 &\geq 35 \\
y_2 + y_3 &\geq 35 \\
y_1, y_2, y_3 &\geq 15
\end{aligned}
$$

8.2 The Element Constraint

The element constraint selects a value to assign to a variable. It can be written

$$
\text{element}(y, (v_1, \ldots, v_n), z) \tag{44}
$$

If the value of y is fixed to i, the constraint fixes z to v_i. If the domain D_y of y is larger than a singleton, the constraint essentially defines a domain for z:

$$
z \in \{v_i \mid i \in D_y\} \tag{45}
$$

The current domain D_z of z can also be used to reduce the domain of y:

$$
y \in \{i \mid v_i \in D_z\} \tag{46}
$$

If the values v_i are numerical, there is an obvious convex hull relaxation for (44):

$$\min\{v_i \mid i \in D_y\} \leq z \leq \max\{v_i \mid i \in D_y\} \tag{47}$$

The element constraint is particularly useful for implementing variable indices, an example of which may be found in the objective function of the traveling salesman problem (42). A constant with a variable index, such as v_y, may be replaced by the variable z and the constraint (44).

An extension of the element constraint implements variable indices for variables:

$$\text{element}(y, (x_1, \ldots, x_n), z) \tag{48}$$

If y is fixed to i, the constraint posts a new constraint $z = x_i$. Otherwise the constraint is essentially a disjunction:

$$\bigvee_{i \in D_y} (z = x_i)$$

Hyperarc consistency can be achieved for (48) as follows [20]. Let D_y, D_{x_i}, D_z be the current domains, and let $\bar{D}_y$, $\bar{D}_{x_i}$ and $\bar{D}_z$ be the reduced domains. Then

$$\bar{D}_z = D_z \cap \bigcup_{i \in D_y} D_{x_i}$$

$$\bar{D}_y = D_y \cap \{i \mid D_z \cap D_{x_i} \neq \emptyset\} \tag{49}$$

$$\bar{D}_{x_i} = \begin{cases} \bar{D}_z & \text{if } \bar{D}_y = \{i\} \\ D_{x_j} & \text{otherwise} \end{cases}$$

Disjunctive relaxations may be derived for (48) using the principles of the previous section. It is proved in [20] that (48) has the following convex hull relaxation when all the variables x_i have the same upper bound m_0:

$$\sum_{i \in D_y} x_i \leq z \leq \sum_{i \in D_y} x_i - (|D_y| - 1)m_0 \tag{50}$$

When each $x_i \leq m_i$ one can write the relaxation

$$\left(\sum_{i \in D_y} \frac{1}{m_i} \right) z \leq \sum_{i \in D_y} \frac{x_i}{m_i} + |D_y| - 1$$

$$\left(\sum_{i \in D_y} \frac{1}{m_i} \right) z \geq \sum_{i \in D_y} \frac{x_i}{m_i} - |D_y| + 1 \tag{51}$$

36

as well as setting $m_0 = \max_i\{m_i\}$ and using relaxation (50) as well.

An expression of the form x_y, where x is a variable, can be implemented by replacing it with z and the constraint (48).

8.3 The Cumulative Constraint

The *cumulative* constraint [1] is used for resource-constrained scheduling problems. Jobs must scheduled so that the total resource consumption at any moment is within capacity.

The constraint may be written

$$\text{cumulative}(t, d, r, L) \tag{52}$$

where $t = (t_1, \ldots, t_n)$ is a vector of start times for jobs $1, \ldots, n$, $d = (d_1, \ldots, d_n)$ is a vector of job durations, $r = (r_1, \ldots, r_n)$ a vector of resource consumption rates, and scalar L the amount of available resources. The domain of each t_j is given as $[a_j, b_j]$, which defines an earliest start time a_j and a latest start time b_j for job j. The cumulative constraint requires that

$$\sum_{\substack{j \\ t_j \le t < t_j + d_j}} r_j \le L, \ \ \text{all } t$$

and $a \le t \le b$.

An important special case is the L-machine scheduling problem, in which each resource requirement r_j is 1, representing one machine. The cumulative constraint requires that jobs be scheduled so that no more than L machines are in use at any time.

The cumulative constraint can be approximated in an MILP formulation if one discretizes time, but this can result in a large number of variables.

A number of domain reduction procedures have been developed for cumulative (e.g., [1, 5, 6, 7, 14, 27]), although none of them are guaranteed to achieve hyperarc consistency. They are generally based on edge-finding ideas.

To create a relaxation, it is convenient to analyze the problem in two parts: the "lower" problem, in which the upper bounds $b_j = \infty$, and the "upper" problem, in which the lower bounds $a_j = -\infty$. Relaxations for the upper and lower problem can then be combined to obtain a relaxation for the entire problem. Only the upper problem is studied here, but the lower problem is closely parallel.

Facet-defining inequalities exist when there are subsets of jobs with the same release time, duration, and resource consumption rate.

Theorem 7 (Hooker and Yan [22]) *Suppose jobs $j_1, \ldots, j_k$ satisfy $a_{j_i} = a_0$, $d_{j_i} = d_0$ and $r_{j_i} = r_0$ for $i = 1, \ldots, k$. Let $Q = \lfloor L/r_0 \rfloor$ and $P = \lceil k/Q \rceil - 1$. Then the following defines a facet of the convex hull of the upper problem.*

$$t_{j_1} + \cdots + t_{j_k} \geq (P+1)a_0 + \frac{1}{2}P[2k - (P+1)Q]d_0 \qquad (53)$$

provided $P > 0$. Furthermore, bounds of the form $t_j \geq a_j$ define facets.

The following valid inequalities are in general non-facet-defining but exist in all problems.

Theorem 8 (Hooker and Yan [22]) *Renumber any subset of jobs $j_1, \ldots, j_k$ using indices $q = 1, \ldots, k$ so that the products $r_q d_q$ occur in nondecreasing order. The following is a valid inequality for the upper problem:*

$$t_{j_1} + \cdots + t_{j_k} \geq \sum_{q=1}^{k} \left((k - q + \tfrac{1}{2}) \frac{r_q}{L} - \tfrac{1}{2} \right) d_q \qquad (54)$$

Suppose for example that there are five jobs and $a = (0, 0, 0, 0, 0)$, $d = (2, 4, 3, 3, 3)$, $r = (4, 2, 3, 3, 3)$, $L = 6$. The minimum makespan is 8, using $(t_1, \ldots, t_5) = (3, 3, 0, 0, 5)$. The following relaxation can be written, where the second constraint is facet-defining and derives from Theorem 7, and the third constraint is from Theorem 8.

$$
\begin{aligned}
\min \quad & z \\
\text{subject to} \quad & z \geq t_1 + 2, t_2 + 4, t_3 + 3, t_4 + 3, t_5 + 3 \\
& t_1 + t_2 + t_3 \geq 3 \\
& t_1 + t_2 + t_3 + t_4 + t_5 \geq 11\tfrac{11}{12} \\
& t_j \geq 0, \quad \text{all } j
\end{aligned}
$$

The optimal value of the relaxation is 5.38, which provides a valid bound $z \geq 5.38$ on the optimal makespan.

9 Conclusion

Table 2 lists the logical-based constraints discussed above and briefly indicates how one might write or reformulate the constraint to achieve consistency, as well as how one might write a tight MILP formulation. Consistency is important for CP-based or CP/MILP hybrid solvers, and it can also help generate tighter MILP formulations. The MILP reformulation is obviously essential if one intends to use an MILP, and it also provides a tight relaxation for hybrid solvers. The situation is summarized in Table 3.

The consistency methods are designed to be applied to groups of constraints and can be implemented either by hand or by the solver. The MILP reformulation would be carried out automatically.

As mentioned at the outset, there are advantages to applying a unified solver directly to logic-based constraints, rather than sending them to MILP solver in inequality form. The advantages are not only computational, but such constraints as all-different, element and cumulative are difficult to formulate in an MILP framework.

The discussion here has focused on obtaining consistency and a tight relaxation before solution starts. Similar methods can be used in the course of solution. CP and hybrid solvers, for example, typically restore hyperarc consistency or some approximation of it whenever domains are reduced by branching or filtering. This is regularly done for all-different, element and cumulative constraints. Resolution could be repeatedly applied to clause sets and to antecedents of conditional constraints in order to maintain full consistency. Cardinality and 0-1 resolution may be too costly to apply repeatedly. It may also be advantageous to update linear relaxations.

Several types of logics other than those discussed here can be given MILP formulations. They include probabilistic logic, belief logics, many-valued logics, modal logics, default and nonmonotonic logics, and if one permits infinite-dimensional integer programming, predicate logic. All of these are discussed in [20]. Many of the resulting MILP formulations provide practical means of reasoning in these various logics. The formulations are omitted here, however, because it is unclear whether it is practical or useful to incorporate them with other types of constraints in a general-purpose model. This remains an issue for further research.

Table 2: Catalog of logic-based constraints.

Type of constraint and examples	To achieve consistency (or ease detection of redundant partial assignments)	To obtain a tight MILP formulation
Formula of propositional logic $a \vee \neg b \vee c$ $a \equiv (b \rightarrow c)$	Convert to clausal form using the algorithm of Fig. 1 or 2 and apply resolution to achieve full consistency, or apply a limited form of resolution.	Convert to clausal form without adding variables (using the algorithm of Fig. 1) and write clauses as 0-1 inequalities. If desired apply cardinality resolution.
Cardinality clauses $\{a, \neg b, c\} \geq 2$	Apply cardinality resolution to achieve full consistency. (Diagonal sums are not needed for consistency but can accelerate checking for redundant assignments.)	Apply cardinality resolution and convert the cardinality clauses to 0-1 inequalities.
0-1 linear inequalities, viewed as logical propositions $3x_1 - 2x_2 + 4x_3 \geq 5$	Apply 0-1 resolution to achieve full consistency (best for small or specially structured inequality sets).	Generate cutting planes.
Cardinality rules $\{a, \neg b, c\} \geq 2$ $\rightarrow \{d, e\} \geq 1$		Generate a convex hull formulation as described in Fig. 6.
Conditional constraints $(b \vee \neg c) \rightarrow (Ax \geq a)$	Consistency unnecessary, but put every antecedent into clausal form.	Use inequality-or global constraints to identify disjunctions of linear systems and relax using a convex hull relaxation (36) or a big-M relaxation given by (33) or (34).
All-different all-different$\{y_1,$ $\ldots, y_n\}$	Use Régin's matching algorithm to achieve hyperarc consistency.	Use the convex hull formulation (43) if it is not too large.
Element element$(y,$ $(v_1, \ldots, v_n), z)$	Use formulas (45) and (46) to achieve hyperarc consistency	Use the range (47).
Extended element element$(y,$ $(x_1, \ldots, x_n), z)$	Use formulas (49) to achieve hyperarc consistency.	Use relaxations (50) and (51).
Cumulative cumulative(t, d, r, L)		Use relaxations (53) and (54).

Table 3: Advantages of consistent and tight formulations.

Type of formulation	Advantage for CP solver	Advantage for hybrid CP/MILP solver	Advantage for MILP solver
Consistent logic-based formulation	Reduces backtracking.	Reduces backtracking.	May help generate tight MILP formulation.
Tight MILP formulation		May provide tight relaxation for better bounds.	Provides tight relaxation for better bounds.

References

[1] Aggoun, A., and N. Beldiceanu, Extending CHIP in order to solve complex scheduling and placement problems, *Mathematical and Computer Modelling* **17** (1993) 57–73.

[2] Balas, E., Disjunctive programming: Cutting planes from logical conditions, in O. L. Mangasarian, R. R. Meyer, and S. M. Robinson, eds., *Nonlinear Programming 2*, Academic Press (New York, 1975), 279-312.

[3] Balas, E., Disjunctive programming, *Annals Discrete Mathematics* **5** (1979): 3-51.

[4] E. Balas, A. Bockmayr, N. Pisaruk and L. Wolsey, On unions and dominants of polytopes, manuscript, http://www.loria.fr/~bockmayr/dp2002-8.pdf, 2002.

[5] Baptiste, P.; C. Le Pape, Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, *Proceedings, UK Planning and Scheduling Special Interest Group 1996* (PLANSIG96), Liverpool, 1996.

[6] Baptiste, P., and C. Le Pape, Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Principles and Practice of Constraint Programming (CP 97)*, Springer-Verlag (Berlin, 1997) 375–89.

[7] Baptiste, P.; C. Le Pape, Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems, *Constraints* **5** (2000) 119–39.

[8] Barth, P., *Logic-Based 0-1 Constraint Solving in Constraint Logic Programming*, Kluwer (Dordrecht, 1995).

[9] Beaumont, N., An algorithm for disjunctive programs, *European Journal of Operational Research* **48** (1990): 362-371.

[10] Berge, C., *Graphes et hypergraphes*, Dunod (Paris, 1970).

[11] Bockmayr, A., and J. N. Hooker, Constraint programming, in K. Aardal, G. Nemhauser and R. Weismantel, eds., *Handbook of Discrete Optimization*, North-Holland, to appear.

[12] Chandru, V. and J. N. Hooker, *Optimization Methods for Logical Inference*, Wiley (New York, 1999).

[13] Chvátal, V., Edmonds polytopes and a hierarchy of combinatorial problems, *Discrete Mathematics* **4** (1973): 305-337.

[14] Caseau, Y., and F. Laburthe, Cumulative scheduling with task intervals, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1996.

[15] Cornuejols, G., and M. Dawande, A class of hard small 0-1 programs, *INFORMS Journal on Computing* **11** (1999) 205-210.

[16] Haken, A., The intractability of resolution, *Theoretical Computer Science* **39** (1985): 297-308.

[17] S. Heipcke, *Combined Modelling and Problem Solving in Mathematical Programming and Constraint Programming*, PhD thesis, University of Buckingham, 1999.

[18] Hooker, J. N., Generalized resolution and cutting planes, *Annals of Operations Research* **12** (1988): 217-239.

[19] Hooker, J. N., Generalized resolution for 0-1 linear inequalities, *Annals of Mathematics and Artificial Intelligence* **6** (1992): 271-286.

[20] Hooker, J. N., *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, Wiley (New York, 2000).

[21] Hooker, J. N., Logic, optimization and constraint programming, *INFORMS Journal on Computing*, to appear.

[22] Hooker, J. N., and Hong Yan, A relaxation of the cumulative constraint, *Principles and Practice of Constraint Programming* (CP02), Lecture Notes in Computer Science **2470**, Springer (Berlin, 2002) 686-690.

[23] Jeroslow, R. E., *Logic-Based Decision Support: Mixed Integer Model Formulation, Annals of Discrete Mathematics* **40**. North-Holland (Amsterdam, 1989).

[24] Milano, M., Integration of OR and AI constraint-based techniques for combinatorial optimization, tutorial, http://www-lia.deis.unibo.it/Staff/MichelaMilano/tutorialIJCAI2001.pdf.

[25] Mitra, G., C. Lucas, S. Moody, Tools for reformulating logical forms into zero-one mixed integer programs, *European Journal of Operational Research* **72** (1994) 262-276.

[26] Nemhauser, G. L., and L. A. Wolsey, *Integer and Combinatorial Optimization*, Wiley (1999).

[27] Nuijten, W. P. M., *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*, PhD Thesis, Eindhoven University of Technology, 1994.

[28] Quine, W. V., The problem of simplifying truth functions, *American Mathematical Monthly* **59** (1952): 521-531.

[29] Quine, W. V., A way to simplify truth functions, *American Mathematical Monthly* **62** (1955): 627-631.

[30] Régin, J.-C., A filtering algorithm for constraints of difference in CSPs, *Proceedings, National Conference on Artificial Intelligence* (1994): 362-367.

[31] Tseitin, G. S., On the complexity of derivations in the propositional calculus, in A. O. Slisenko, ed., *Structures in Constructive Mathematics and Mathematical Logic, Part II* (translated from Russian, 1968), 115-125.

[32] Williams, H. P. *Model Building in Mathematical Programming*, 4th ed., Wiley (Chichester, 1999).

[33] Williams, H. P., and J. M. Wilson, Connections between integer linear programming and constraint logic programming–An overview and introduction to the cluster of articles, *INFORMS Journal on Computing* **10** (1998) 261-264.

[34] Williams, H.P., and Hong Yan, Representations of the all different predicate of constraint satisfaction in integer programming, *INFORMS Journal on Computing* **13** (2001) 96-103.

[35] Wilson, J. M., Compact normal forms in propositional logic and integer programming formulations, *Computers and Operations Research* **90** (1990): 309-314.

[36] Wolsey, L. A., *Integer Programming*, Wiley (1998).

[37] Yan, H., and J. N. Hooker, Tight representation of logical constraints as cardinality rules, *Mathematical Programming* **85** (1999): 363-377.