Mixed Global Constraints and Inference in Hybrid CLP–IP Solvers

Greger Ottosson [a], Erlendur S. Thorsteinsson [b], John N. Hooker [b]

[a] *Department of Information Tech./Computing Science; Uppsala University; PO Box 311; S-751 05 Uppsala; Sweden.*

[b] *Graduate School of Industrial Administration; Carnegie Mellon University; Pittsburgh, PA 15213; U.S.A.*

Abstract

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

1 Introduction

In this paper we continue exploring the integration of constraint programming and mathematical programming, specifically Constraint Propagation (CP) and Linear Programming (LP), extending our previous work [9,10,11,12,13]. In particular, we examine in more detail how to model for a hybrid solver and how to solve hybrid models, inter alia by giving specific examples. We also provide benchmarks for a production planning problem. The main contributions of this paper are:

- Mixed global constraints connecting CP and LP:
  - Variable subscripts and the compilation of variable subscripts in continuous functions, i.e., an extension of the `element` constraint to the continuous domain. We also describe its relation to disjunctive programming.

  - A mixed global constraint for semi-continuous piecewise linear functions.
- A scheme for bidirectional inference between CP and LP, i.e., between the Finite Domain constraint store (FD store) and the Linear Programming constraint store (LP store) [12].
- New search strategies for hybrid models.

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

The key decisions to be made for integrating Constraint (Logic) Programming (CLP) and Integer Programming (IP) are the (a) models, (b) inference, (c) relaxations, and, (d) search and branching strategies to use. For example, [3] introduces a framework with a combined constraint store and symbolic constraints that produce cutting planes; [8] combines two different models in two synchronized search trees; and, [16] automatically produces and updates a shadow copy of a CLP model on the continuous side, with constraint propagation and linear relaxations in a single search tree. A common feature of these methods is to communicate bounds, as introduced in [2].

In [12], we advocate to use neither the CLP nor the IP model but rather to model specifically for the hybrid solver, roughly separating the problem into a discrete part (FD store) and a continuous part (LP store). This achieves domain reduction on the FD store (constraint propagation), inference from the FD store to the LP store (bounds and cutting planes) and a natural relaxation (the LP relaxation).

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

This paper is organized as follows. This section laid out the history of efforts in integrating CLP and IP. Section 2 introduces our framework, Mixed Logical/Linear Programming (MLLP). In Sec. 3 we expand this framework with mixed global constraints, taking variable subscripts and piecewise linear functions as specific examples. Section 4 focuses on algorithms and rules for inference and branching strategies. In Sec. 5 we introduce a production planning problem and then compare MLLP computationally with other approaches in Sec. 5.3. Finally, Sec. 6 summarizes our results.

2 Mixed Logical/Linear Programming (MLLP)

To lay the basis for the subsequent discussion, we recapitulate the basic framework of MLLP, proposed in [9,10,11,12,13]. In that framework, constraints are in the form of conditionals that link the discrete and continuous elements of the problem. An MLLP model has the form

$$\begin{aligned} \min\ & cx \\ \text{s.t.}\ & h_i(y) \rightarrow A^i x \geq b^i, \quad i \in I, \\ & x \in \mathbb{R}^n,\ y \in D, \end{aligned} \tag{1}$$

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

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

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

3 Mixed Global Constraints

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

Mathematical programming has still not seen the advantage of global con-

straints to the same extent. Modeling languages for LP/IP, such as AMPL [7], do include some constructs to aid the modeler in writing compact and easy-to-understand models. Those constructs, however, have to be transformed into linear inequalities before being sent to an LP/IP solver and the structural information is lost in the process. Although the structure may be known to the modeler it can not be communicated to the solver and the solver has to find and recognize the structure on its own to be able to apply logical processing to it, resulting in less efficient solution algorithms than what would have been possible. Recently though some work has been done on this topic [3].

In a framework such as MLLP, *mixed* global constraints serve both as a modeling tool and a way to exploit structure in the solution process. Mixed global constraints can be written in the form (1) as conditionals, analogous to global constraints in CLP, but improve the solution process by improving the propagation. This will be illustrated for variable subscripts and piecewise linear functions.

3.1 Variable Subscripts in Linear Constraints

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

A variable subscripted expression can, in principle, be written in the more primitive form (1) of conditional constraints. For example, the constraint $z \geq \sum_j c_{jy_j}$ can be written $z \geq \sum_j z_j$, if the following conditional constraints are added to the model,

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

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

A constraint called (discrete) `element`, which mimics array lookup, has been used for a long time in the CLP community to represent variable subscript [14]. The expression $z = x_y$, where z and y are discrete variables and $x_1, \ldots, x_n$ are discrete variables (or constants), is equivalent to

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

given that the domain of y is $D_y = \{1, \ldots, n\}$. Propagation for this constraint

is usually [hyper]arc- or bounds-consistency if x is an array of constants [variables].

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

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

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

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

$$\begin{array}{ll}
\texttt{element}(y, [x_1, \ldots, x_n], z'_1), & \texttt{element}(y, [c_{11}, \ldots, c_{1n}], w_1), \\
\texttt{element}(y, [d_1, \ldots, d_n] \times z'_1, z_1), & \texttt{element}(y, [c_{21}, \ldots, c_{2n}], w_2), \\
\texttt{element}(y, [x_1, \ldots, x_n], z'_2), & \cdots \\
\texttt{element}(y, [d_1, \ldots, d_n] \times z'_2, z_2), & \texttt{element}(y, [c_{m1}, \ldots, c_{mn}], w_m).
\end{array}$$

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

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

$$\begin{array}{ll}
z + w = 42, \quad z \geq 0, & \\
\texttt{element}(y, [x_1, \ldots, x_n], z'), & (2) \\
\texttt{element}(y, [d_1, \ldots, d_n] \times z', z), & (3) \\
\texttt{element}(y, [\textstyle\sum_{i \in S} c_{i,1}, \ldots, \sum_{i \in S} c_{i,n}], w). & (4)
\end{array}$$

As illustrated in the example above, our MLLP modeling language and solver

support three kinds of subscripted expressions in linear functions:

$$\texttt{element}(y, [a_1, \ldots, a_n], z) \quad \Leftrightarrow z = a_y, \tag{5}$$
$$\texttt{element}(y, [x_1, \ldots, x_n], z) \quad \Leftrightarrow z = x_y, \tag{6}$$
$$\texttt{element}(y, [a_1, \ldots, a_n] \times x, z) \Leftrightarrow z = a_y x. \tag{7}$$

The modeler compiles general expressions containing variable subscripts by summing and chaining together (5)–(7), and the solver can propagate bounds and create cuts for these structures. We observe that (a) any linear expression with variable subscripts can be decomposed by the modeler using these three forms, and, (b) they are simple enough to propagate efficiently [13].

Note that subscript folding is limited by this vocabulary, subscripted variables can, e.g., not be folded. Also, some information may be lost when decomposing an expression into several `element` constraints. For example, the minimum lower bound for z' in our example above, derived using bounds propagation on (2), might occur at a different index than the minimum of the d_i's in (3). This decoupling might thus give a weaker bound on z than by indexing directly on $d_y x_y$ in a single `element` constraint. There would not be any benefit, however, from having a new variant of the `element` constraint for $d_y x_w$ (except in the special case of $y \equiv w$ as in the example above) as $d_y x_w$ is naturally decoupled by the two different indexing variables and no information is lost by decomposing $z = d_y x_w$ to $z' = x_w$ and $z = d_y z'$.

More complex expressions can be handled by a more general form of the `element` constraint, instead of using decomposition. As we alluded to in Sec. 1, there is an intimate connection between variable subscripts in linear expressions (the mixed `element` constraint) and disjunctive programming. In fact, the three forms of the mixed `element` constraint above, eq. (5)–(7), are special cases of a general mixed global disjunctive constraint

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

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

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

which is far more complex than the inference for the three frequently and naturally occurring structures (5)–(7), although it has been studied in the literature [1].

The propagation rules for the mixed `element` constraints are similar to the discrete case. Let the interval $[\min(x_i), \max(x_i)]$ be defined by the bounds on

x_i. Then upon change of the domain of y, we can compute $\min = \{\min(x_i) \mid i \in D_y\}$ and $\max = \{\max(x_i) \mid i \in D_y\}$ and add new bounds $\min \leq z \leq \max$ to the LP. Reversely, the bounds of z can be used to prune D_y. More on propagation and relaxations for the `element` constraint can be found in [13].

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

3.1.1 Variable Subscripts in Bounds

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

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

$$l_y \leq x \leq u_y \tag{8}$$

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

$$z_1 \leq x, \quad x \leq z_2, \tag{9}$$
$$\texttt{element}(y, [l_1, \ldots, l_n], z_1), \tag{10}$$
$$\texttt{element}(y, [u_1, \ldots, u_n], z_2), \tag{11}$$

which introduces two new variables, two mixed `element` constraints and two linear inequalities. Secondly, and a more refined trap for our solver, the values of x, z_1 and z_2 in a continuous solution may be such that no value $i \in D_y$ satisfies (10)–(11), despite the fact that some value $i \in D_y$ together with x satisfies the original variable subscripted bounds (8). Methods aimed at finding such satisfying values (see Sec. 4.1) might then fail unnecessarily. This happens in practice and impedes the solver in completing the solution and causes extra branching in the search.

These two problems are avoided by compiling the bounds directly into

$$\texttt{element}_{\leq}(y, [l_1, \ldots, l_n], x), \tag{12}$$
$$\texttt{element}_{\geq}(y, [u_1, \ldots, u_n], x), \tag{13}$$

where (12) [(13)] only performs lower [upper] bound propagation. Since the artificial variables are eliminated, values for y satisfying (12)–(13) will be found correctly. The mixed `element` constraint described above is declaratively equivalent to $\texttt{element}_{\leq} \wedge \texttt{element}_{\geq}$, and is referred to as $\texttt{element}_{=}$ in ambiguous cases.

3.2 Semi-continuous Piecewise Linear Functions

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

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

Using variable subscripts, we can model the function as

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

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

$$v = z_1 + z_2 \tag{14}$$
$$\texttt{element}_{=}(y, [c_1, \ldots, c_n] \times u, z_1) \tag{15}$$
$$\texttt{element}_{=}(y, [\underline{v}_1 - c_1\underline{u}_1, \ldots, \underline{v}_n - c_n\underline{u}_n], z_2) \tag{16}$$
$$\texttt{element}_{\leq}(y, [\underline{u}_1, \ldots, \underline{u}_n], u) \tag{17}$$
$$\texttt{element}_{\geq}(y, [\bar{u}_1, \ldots, \bar{u}_n], u) \tag{18}$$
$$\texttt{element}_{\leq}(y, [\underline{v}_1, \ldots, \underline{v}_n], v) \tag{19}$$
$$\texttt{element}_{\geq}(y, [\bar{v}_1, \ldots, \bar{v}_n], v) \tag{20}$$

Consider the piecewise linear function in Fig. 1. It consists of four segments (numbers in circles), the first is the origin (of zero-width), the next three range

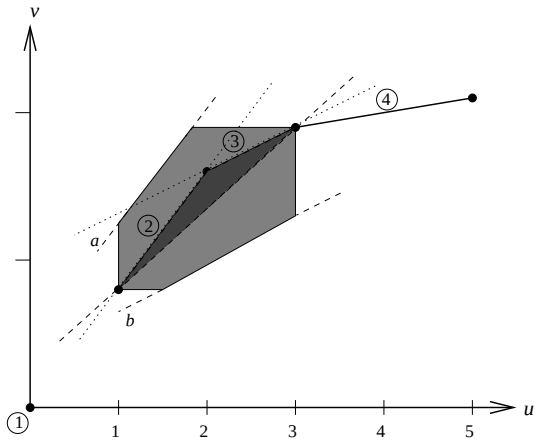

Figure 1. A piecewise linear function. The light grey area depicts the linear relaxation using variable subscripts, the dark grey area the relaxation obtained with the `piecewise` constraint.

between u-values 1–2, 2–3 and 3–5. At the depicted state the segment chosen is constrained to be 2 or 3, i.e., $y \in \{2, 3\}$. Bounds propagation in the `element` constraints will now produce the linear relaxation shown in light grey. The horizontal and vertical borders are obtained by updating the bounds for the continuous variables u and v, eq. (17)–(20). The other two borders (marked a and b) have slopes which are the effects of the bounds propagation of (15), and an offset which comes from (16). Note that the upper bound a is parallel to the steeper segment 2 and the lower bound b is parallel to segment 3.

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

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

where y is the discrete variable indicating the segment in which the values of u and v lie, and O indicates the orientation of the constraint, i.e., whether v has to be above, below or on the piecewise linear function. Then a tighter linear relaxation can be produced by instead adding the cuts surrounding the dark grey triangle in Fig. 1. If O is above [below] only the lower [upper] cuts have to be posted, if it is on then both the lower and the upper cuts have to be posted. In addition to posting and updating the cuts described, the constraint performs bounds propagation on both the discrete and the continuous variables. For $D_y = \{p, \ldots, q\}$ the propagation rules for concave functions are:

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

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

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

Declaratively, the `piecewise` constraint can be written in the form (1) as follows,

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

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

The general syntax of the `piecewise` constraint, i.e., specifying both the start and end point of each segment, is necessary due to semi-continuous characteristic of a general piecewise linear function. Note, e.g., that a fixed-charge piecewise linear constraint is a special case of this constraint. Furthermore, the syntax allows for functions that are neither convex nor concave, and generating the convex hull for such general semi-continuous piecewise functions is then a matter of finding the convex hull of a set of points. This is a well known problem in computational geometry and can be done efficiently [4]. A simpler syntax for continuous functions could of course be adopted, e.g., as in AMPL [7] and OPL [17], where the functions are assumed to be continuous and starting at the origin, and only the end point and slope of each segment are specified.

4 Algorithmic Extensions

In CLP, good support for defining problem specific search strategies is essential. Strategies derive branching decisions from the current domains of variables and the structure of the constraint graph. Well-known strategies are *fail-first*, *most-constrained* and *earliest-start-time* (scheduling problems) for variable selection and *domain-splitting* for value choice [14].

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

The basic strategies from CLP remain available for a hybrid system like MLLP since the discrete decision variables still have domains and the discrete constraints still form a constraint graph. But in a solver where discrete and continuous variables have been separated, like in MLLP, discrete variables will not readily have values from the relaxation. The relaxation provides a value for x in (1) and a part of y might be determined by branching, but y is not given a value by the relaxation since it is not a part of it.

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

4.1 Back-Propagation

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

At this point, if all constraints are determined, we have a complete solution. If not, we might need to branch further. However, it might possible to extend the solution x^* of the relaxation to a complete solution (y^*, x^*) that satisfies all constraints. And if not, we should deduce information for branching strategies.

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

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

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

- For any

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

  if:

  $O \in \{\textsf{below}, \textsf{on}\}$: let $D_y = D_y \cap \{j \mid (\underline{u}_j \leq u^* \leq \bar{u}_j) \wedge (v^* \leq v(u^*))\}$,

  $O \in \{\textsf{above}, \textsf{on}\}$: let $D_y = D_y \cap \{j \mid (\underline{u}_j \leq u^* \leq \bar{u}_j) \wedge (v^* \geq v(u^*))\}$.

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

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

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

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

In [6] a similar scheme called *unimodular probing*, is used to derive constraint violations from a totally unimodular subset of linear inequalities solved as an LP. The violated constraints indicates possible branching choices. Similarly, the value extraction for *shadowed* variables, i.e., a variable present in both CP and LP [16,17], is also a special case of back-propagation, allowing branching strategies based on the LP solution values.

4.2 Branching Strategies

In a branching strategy there are two choices to make; what *variable* to branch and on, and what *value* to set it to. In IP it is very common to branch on the

most fractional variable, resolving the "biggest" inconsistencies first. Using the domains after back-propagation, we can accomplish something similar, called *back-propagation-failure*. This strategy selects first any variable whose domain became empty during back-propagation, and secondly any variable whose domain after back-propagation was not singleton, and finally falls back on a standard fail-first strategy on the domains as they were before back-propagation. Variables with singleton domains after back-propagation are skipped since they have a consistent value.

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

All of the above value heuristics can also be combined with a *best-branch* selection, i.e., the objective value for the branches are computed, and the branches are tried in the order of greatest potential (best objective value). Computing the objective value before choosing a branch should not be confused with *strong-branching* [5]. Strong branching precomputes objective values for use in *variable* selection and will therefore incur overhead by solving relaxations for branches never taken. In contrast, *best-branch* is a *best-bound* node selection strategy, where the set of candidate nodes are the children of the current node. It will only move some computation up in the search tree, since all branches will eventually be considered, and thus not cause any overhead.

5 A Production Planning Problem

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

5.1 An MLLP Model

Using the piecewise global constraint described in Sec. 3.2, the formulation is straightforward. Variable r_j is the revenue for product j, c_i the cost of resource i, y_j^R the segment where production v_j lies for product j, y_i^C the segment where usage u_i lies for resource i, and s_j is the production scale for product j. Parameters $\underline{u}_{ik}$ and $\bar{u}_{ik}$ denote upper and lower range of segment k for resource i, and $\underline{C}_{ik}$ and $\bar{C}_{ik}$ the corresponding accumulated cost at those points. Similarly, $\underline{v}_{jk}, \bar{v}_{jk}$ and $\underline{R}_{jk}, \bar{R}_{jk}$ describe the piecewise linear revenue function. Note that $\underline{u}_{i1} = \bar{u}_{i1} = 0$ and $\underline{u}_{i2} > 0$, enforcing the minimum purchase quantity directly through the global semi-continuous piecewise constraint.

$$
\begin{aligned}
\max \quad & \sum_j r_j - \sum_i c_i \\
\text{s.t.} \quad & \texttt{piecewise}(y_i^C, u_i, \underline{u}_{i1}, \ldots, \underline{u}_{in}, \bar{u}_{i1}, \ldots, \bar{u}_{in}, \\
& \qquad\qquad c_i, \underline{C}_{i1}, \ldots, \underline{C}_{in}, \bar{C}_{i1}, \ldots, \bar{C}_{in}), && \forall i, && (21) \\
& \texttt{piecewise}(y_j^R, v_j, \underline{v}_{j1}, \ldots, \underline{v}_{jm}, \bar{v}_{j1}, \ldots, \bar{v}_{jm}, \\
& \qquad\qquad r_j, \underline{R}_{j1}, \ldots, \underline{R}_{jm}, \bar{R}_{j1}, \ldots, \bar{R}_{jm}), && \forall j, && (22) \\
& \sum_j a_{ij} v_j \leq u_i, && \forall i, && (23) \\
& \sum_j v_j \leq plant\_cap, && && (24) \\
& \underline{w}_{j,s_j} \leq v_j \leq \bar{w}_{j,s_j}, && \forall j, && (25) \\
& u_i, c_i, v_j, r_j \geq 0, && \forall i, j, \\
& y_i^C \in \{1, \ldots, n\},\ y_j^R \in \{1, \ldots, m\},\ s_j \in \{1, \ldots, l\}, && \forall i, j.
\end{aligned}
$$

Note the variable subscripted bounds (25) enforcing the restrictions on the scale of production.

5.2 Other Models

An IP Model: A traditional IP model, see Fig. 2, can be found by expanding the `piecewise` constraints to LP constraints for the revenue (concave maximization (`below`), constraints (26)–(27)), where $\hat{R}_{jk}$ is the revenue of each unit produced in interval k, and MIP constraints for the cost (concave minimization (`above`), constraints (28)–(31)). Due to lack of variable subscripts, the production scale intervals have to be encoded as a MIP structure (constraints (32)–(34)).

Notice that y_{jk} are SOS–3 variables [5] due to (34), which can be used by the IP solver. Informing the IP solver about the SOS–3 variables is optional, the model is still valid without that information. It is necessary, however, to tell

$$
\begin{aligned}
\max\ & \sum_{j,k} \hat{R}_{jk} v_{jk} - \sum_i \left(\underline{C}_{i2} \lambda_{i1} + \sum_{k \geq 2} \bar{C}_{ik} \lambda_{ik} \right) \\
\text{s.t.}\ & v_j = \sum_k v_{jk}, && \forall j, && (26) \\
& v_{jk} \leq \bar{v}_{jk} - \underline{v}_{jk}, && \forall j,k, && (27) \\
& u_i = \underline{u}_{i2} \lambda_{i1} + \sum_{k \geq 2} \bar{u}_{ik} \lambda_{ik}, && \forall i, && (28) \\
& \sum_k \lambda_{ik} = 1, && \forall i, && (29) \\
& \{\lambda_{i1}, \ldots, \lambda_{in}\} \text{ is an SOS–2 set}, && \forall i, && (30) \\
& \underline{u}_{i2} b_i \leq u_i \leq M b_i, && \forall i, && (31) \\
& v_j = \sum_k w_{jk}, && \forall j, && (32) \\
& \underline{w}_{jk} y_{jk} \leq w_{jk} \leq \bar{w}_{jk} y_{jk}, && \forall j,k, && (33) \\
& \sum_k y_{jk} = 1, && \forall j, && (34) \\
& \sum_j v_j \leq plant\_cap, && && (35) \\
& \sum_j a_{ij} v_j \leq u_i, && \forall i, && (36) \\
& v_j, v_{jk}, w_{jk}, u_i, \lambda_{ik} \geq 0, && \forall i,j,k, && \\
& y_{jk} \in \{0,1\}, && \forall i,j. &&
\end{aligned}
$$

Figure 2. An IP model for the production planning problem.

the solver about the SOS–2 variables, λ_{ik}, as the model is incorrect without that information.

Finally, note that the semi-continuous minimum purchase quantity is enforced through an extra constraint (31).

An OPL Model: The modeling language OPL [17], like MLLP, allows formulations which mix discrete and continuous constraints. Variable subscripts are supported, but only for discrete variables or constants and not in continuous linear constraints nor in bounds. There is also a construct for modeling piecewise linear functions, but this construct is merely syntactic sugar for an LP or MIP structure, resolved at compile time (same as in AMPL [7]). Furthermore, the piecewise construct does not allow semi-continuous functions.

Nonetheless, the problem can be stated in a fairly compact manner in OPL, see Fig. 3. The variable subscripts in the bounds, enforcing the restrictions on the scale of the production, are set on integer variables v'_j which are shadow copies of the variables v_j in the LP (constraints (39)–(40)).

$$\max \left(\sum_j \texttt{piecewise}\{\hat{R}_{jk} \to \bar{v}_{jk};\ 0\}\ v_j \right) - \left(\sum_i \texttt{piecewise}\{\hat{C}_{ik} \to \bar{u}_{ik};\ M\}\ u_i \right)$$

$$\text{s.t. } \underline{u}_{i2} b_i \le u_i \le M b_i, \qquad \forall i, \tag{37}$$

$$\sum_j a_{ij} v_j \le u_i, \qquad \forall i, \tag{38}$$

$$\underline{w}_{j,s_j} \le v'_j \le \bar{w}_{j,s_j}, \qquad \forall j, \tag{39}$$

$$v_j = v'_j, \qquad \forall j, \tag{40}$$

$$\sum_j v_j \le plant\_cap, \tag{41}$$

$$v_j, u_i \ge 0, \qquad \forall i, j,$$

$$v'_j \in \{0, \ldots, \infty\}, \qquad \forall j,$$

$$s_j \in \{1, \ldots, l\}, \qquad \forall j,$$

$$b_i \in \{0, 1\}, \qquad \forall i.$$

Figure 3. An OPL model for the production planning problem.

Unless specified, OPL applies a default branching strategy, which we assume varies with the problem, but whether it is in fact so is not documented in the OPL manual [17]. We tried the following strategy, trying to to some extent simulate the effect of back-propagation of MLLP:

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

However, benchmarking (Sec. 5.3) showed that the default strategy consistently performed better than what we could achieve with this strategy.

Note that the OPL is data dependant in the same way as the IP model. Both assume that the piecewise linear functions are continuous and thus the minimum purchase quantity has to be modeled specifically. In the OPL model it is enforced through an extra constraint (37).

5.3 Benchmarks

In this section we evaluate the performance of three solvers on these three models. We used the same data sets for the different models; each number

Table 1
Benchmark results for the MLLP model.

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

Table 2
IP model, default settings.

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

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

Superscripted digits indicates the number (out of 10 problems) that were solved to proven optimality within 100 000 nodes (if at least one was solved), '–' indicates that none of the ten instances could be solved. 'Nodes' is the number of nodes required to prove optimality, and 'Opt' is the node in which the optimal solution was found, on average. All numbers include the problems that were not solved to optimality, i.e., the 100 000 nodes and the time[1] it took to process them are weighed in.

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

Table 3
IP model, depth-first search and maximum infeasibility variable selection.

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

Table 1 shows the results obtained with our MLLP solver . We branch on the piecewise intervals and the production scales simultaneously, selecting variables according to *back-propagation-failure* and values using *triple-on-lp* (see Sec. 4.2 for details).

Table 2 shows the performance of CPLEX 6.0.1 with default settings on the IP model. 'Bool.' indicates the number of 0–1 variables in the problem, 'PP' stands for "after preprocessing", and 'NZ' indicates the number of nonzero coefficients in the LP matrix. CPLEX uses best-bound node selection by default, i.e., it does not perform a depth first search. For sake of comparison with MLLP and OPL, Tab. 3 shows benchmarks results for CPLEX with depth-first search and maximum infeasibility variable selection.

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

Table 4 shows the performance of OPL. The number of rows and columns is after the piecewise constructs have been expanded to LP and MIP variables and constraints.

Table 4
OPL model, default settings

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

5.3.1 Comments on Computational Results

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

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

6 Conclusion

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

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

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

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

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

References

[1] E. Balas. Disjunctive programming. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II, 5*, pages 3–51, Amsterdam, 1979. Annals of Discrete Mathematics, North-Holland.

[2] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 8, pages 245–272. Elsevier, 1995.

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

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[5] CPLEX. *CPLEX Manual*, 1998. URL http://www.cplex.com.

[6] H. El Sakkout, T. Richards, and M. Wallace. Minimal perturbance in dynamic scheduling. In Prade [15], pages 504–508.

[7] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL – A Modeling Language for Mathematical Programming.* The Scientific Press, South San Francisco, 1993.

[8] S. Heipcke. Integrating constraint programming techniques into mathematical programming. In Prade [15], pages 259–260.

[9] J. N. Hooker. Logic-based methods for optimization. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

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

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

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

[13] John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson, and Hak-Jin Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review, special issue on AI/OR*, 2000.

[14] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[15] Henri Prade, editor. *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*. John Wiley & Sons, 1998.

[16] Robert Rodošek, Mark Wallace, and Mozafar Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Baltzer Journals*, 1997.

[17] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.