A Unit Two Variable Per Inequality Integer Constraint Solver for Constraint Logic Programming

Warwick Harvey and Peter Stuckey
Department of Computer Science
University of Melbourne
Parkville 3052
Email: {warwick,pjs}@cs.mu.OZ.AU
Tel: {+61-3-9287-9171, +61-3-9287-9155}
Fax: +61-3-9348-1184
http://www.cs.mu.OZ.AU/{~warwick,~pjs}

Abstract

The traditional approach to handling integer problems in constraint logic programming systems is to use propagation-based finite domain solvers. These solvers are weak (they do not detect unsatisfiability of complex integer constraints in many cases), but efficiently implementable. They usually require explicit upper and lower bounds for each of the variables of the problem, and if these bounds are too large they can be very inefficient. We present an alternative approach to solving integer constraints based on a polynomial-time solver for a restricted class of integer constraints, constraints of the form $ax + by \leq d$ where $a, b \in \{-1, 0, 1\}$. We use this solver as a basis for a complete integer solver, by combining it with a propagation-based solver. The resulting system improves the propagation approach on a number of different classes of problem.

Key Words: constraint logic programming, linear integer constraints, constraint solving algorithms

1 Introduction

Integer constraints are an important class of constraints used to represent many forms of problems of practical interest, for example scheduling, resource allocation and route planning. The problem of determining satisfiability of linear integer constraints is NP-complete. Hence any CLP system that handles linear integer constraints must be either incomplete (that is, not always able to determine the answer) or inefficient. For practical purposes incompleteness seems to be the only choice as complete integer solvers are too slow — and indeed most CLP systems wish to handle non-linear integer constraints in some form, which makes the satisfiability question undecidable.

For a solver in the context of a CLP system the tradeoff between efficiency and degree of completeness is a subtle one: a more complete solver can drastically reduce search space of a goal by detecting failure earlier, so that overall a solution is found faster. Alternatively the extra overhead of a more complete solver may be significant, and the search space not markedly reduced by its extra power, hence a solution may be found much slower.

The current method of choice for solving integer constraints in CLP systems is based on propagation. Such solvers are very efficient and relatively incomplete. Propagation solvers are typically complete only for the case that each constraint involves a single variable. Call this the class of (linear) single-variable-per-inequality (SVPI) integer constraints. Propagation solvers work by extracting SVPI information from the interaction of non-SVPI constraints with SVPI information.[1] One can consider them as a (very simple) complete constraint solver for SVPI constraints augmented with a propagation mechanism. Usually they require that each variable has an explicit upper and lower bound, and in practice they are usually coupled with an enumeration strategy in the program goal so that by the end of a derivation each variable is fixed to a unique value. Once every variable is fixed the propagation solver again can answer completely (*true* or *false*) by checking that these values satisfy every constraint.

Constraint logic programming (CLP) has successfully tackled many integer problems (e.g. [8]), using solvers based on a propagation approach. These methods usually work very well when all variables involved have natural bounds, but problems can arise if this is not the case, for instance in some temporal reasoning problems. When there is no natural upper or lower bound for a variable, the standard approach has been to choose an ad-hoc bound. The problem is that such a bound could be too tight (leading to the exclusion of feasible solutions), or too loose (leading to inefficiency). Another problem with bounds propagation based techniques is the way they discover transitive consequences. For instance, consider $x + 1 \leq y \leq x$, with $0 \leq x \leq k, 0 \leq y \leq k$. Clearly this system is unsatisfiable, by a simple transitivity argument. However, detecting this by bounds propagation requires cost proportional to k. Note that this problem is compounded if good, natural bounds are not available.

In order to provide a stronger constraint solver an obvious approach is to consider the class of (linear) two-variable-per-inequality (TVPI) integer constraints. Determining the satisfiability of real TVPI constraints is strictly less difficult than for arbitrary real linear inequalities, but this is not the case for integers. Integer TVPI problems are, surprisingly, NP-complete [5]. Hence we need to restrict the class of constraints even further if we desire a practical solver.

A class of constraints intermediate between SVPI and TVPI was introduced in [4]. This is the class of TVPI constraints $ax + by \leq d$ with unit coefficients, that is, $a, b \in \{-1, 0, 1\}$. We call these *unit TVPI constraints* or *UTVPI constraints*. UTVPI constraints are expressive enough to describe all constraints occurring in many scheduling and temporal reasoning problems. And for many other problems UTVPI constraints make up the majority of constraints occurring in the problem description. Analogous to the bound propagation approach we can begin from a complete UTVPI solver and add propagation methods to handle more complex constraints.

In this paper we give a direct mathematical formulation of a constraint solver for integer UTVPI constraints based on transitive closure. As opposed to the algorithm in [4], in our formulation tightening and transitive closure are performed in a single step. We show that this algorithm produces a set of UTVPI constraints closed under transitivity and tightening. We describe how this can be used as the basis for constraint solvers able to handle larger classes of constraints. In particular we show how it is usable as the basis of a solver for arbitrary integer constraints. We give empirical results showing the advantages and disadvantages of this form of constraint solver for solving integer problems.

The remainder of the paper is organized as follows. In the next section we briefly introduce constraint logic programming. Then in Section 3 we define the UTVPI constraint solver, and in Section 4 we discuss extensions to handle larger classes of constraints. We give empirical results comparing the UTVPI solver to propagation-based approaches in Section 5, and finally in Section 6 we conclude.

[1] They usually also do the same with single-variable-per-disequality (SVPD) constraints, but we ignore this for the purposes of this discussion. Disequality constraints are discussed further in Section 4.

2 Constraint Logic Programming

Constraint logic programming is a generalisation of traditional logic programming, where the unification operation has been replaced by constraint solving over some domain (for example the reals, or the integers). What follows is a brief introduction to CLP, biased towards integer solvers. A more comprehensive introduction can be found in [2].

A *primitive constraint* is any fundamental constraint handled by the constraint solver. For example, $X \geq 2$ and $Y = X + 1$ are primitive (integer) constraints. A *constraint* is a (possibly empty) conjunction of primitive constraints. A *solver*, $solv(c)$ maps a constraint c to one of *true*, *false*, or *unknown*. A value of *true* implies that c is satisfiable, a value of *false* implies that c is unsatisfiable, while a value of *unknown* means the solver does not know whether c is satisfiable or not. Most CLP systems which are based on incomplete solvers (i.e. solvers that can return *unknown*) treat answers of *true* and *unknown* identically, and so they are often not distinguished.

An *atom*, A, is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary *predicate symbol* and $t_1, \ldots, t_n$ are terms from the constraint domain. A *literal*, L, is then either an atom or a primitive constraint. A sequence of literals, $L_1, L_2, \ldots, L_m$, where $m \geq 0$, is called a *goal*, G. If $m = 0$, the goal is said to be *empty*, and is written as $\Box$. A *rule*, R, is of the form A :- B, where A is an atom and B is a goal. A is called the *head* of the rule, and B is called the *body*. A *program*, P, is just a set of rules.

A *state* is a pair written $\langle G|c \rangle$, where G is a goal and c is a constraint. A goal is executed with respect to a program by using derivations mapping one state to another. A *derivation step* from $\langle G_0|c_0 \rangle$ to $\langle G_1|c_1 \rangle$, written $\langle G_0|c_0 \rangle \Rightarrow \langle G_1|c_1 \rangle$, is defined as follows. Let G_0 be of the form $L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_m$, where the L_j are literals. Let L_i be the *selected literal* for the derivation step. If L_i is an atom, c_1 is c_0, and G_1 is obtained by rewriting G_0 at L_i based on (a renamed version of) some rule R in the program. The rewriting proceeds as follows. Let L_i be of the form $p(t_1, \ldots, t_n)$, and let R be of the form $p(s_1, \ldots, s_n)$:- B, where the variables of R are disjoint from those of G_0. Then G_1 is

$$L_1, \ldots, L_{i-1}, t_1 = s_1, \ldots, t_n = s_n, B, L_{i+1}, \ldots, L_m$$

If the selected literal L_i is not an atom but a primitive constraint, then c_1 is $c_0 \wedge L_i$, and G_1 depends on the consistency of c_1. If $solv(c_1) = false$, then G_1 is the empty goal $\Box$; otherwise G_1 is $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$.

Operationally, a goal G is "executed" by starting in the state $\langle G|true \rangle$ and performing derivation steps until the current goal is empty. If the resulting constraint c_n might have a solution (i.e. $solv(c_n) \in \{\ true,\ unknown\ \}$), then the derivation is considered successful, with c_n being the *answer constraint*. Note that c_n may not be satisfiable (when $solv(c_n) =$ *unknown*).

Otherwise (i.e. $solv(c_n) = false$), the computation *backtracks* to the previous state, a new (different) rule is selected for the derivation step, and the computation proceeds from there. If there are no more rules to try for the selected literal at a given state, then the computation continues to backtrack until it reaches a state where there is such a choice, or the initial state has been reached, in which case the derivation has failed.

3 Unit TVPI Constraints

Unit two-variable-per-inequality (UTVPI) constraints are expressive enough to capture the constraints involved in scheduling problems. For example consider the project represented

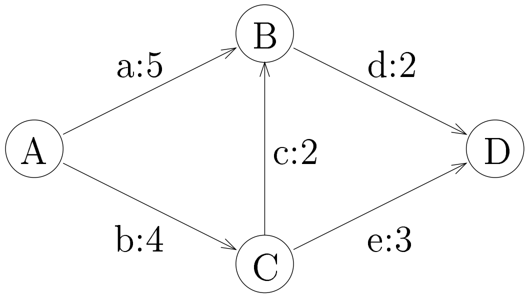

Figure 1: A simple scheduling problem

in Figure 1, where each arc is a task that requires a certain duration to be performed. Each node in the graph represents a constraint on the times the relevant jobs can start. For example, if we denote the start time of job v by S_v, the constraint represented by node X is:

$$S_d \geq S_a + 5 \quad \wedge \quad S_d \geq S_c + 2$$

(job d must start no earlier than the end of a, and no earlier than the end of c).

We can also represent resource constraints, such as if tasks c and d require the same worker, and are thus not allowed to overlap:

$$S_d \geq S_c + 2 \quad \vee \quad S_c \geq S_d + 3$$

One of the problems with solvers based on bounds propagation is that the discovery that

$$S_d \geq S_c + 2 \quad \wedge \quad S_c \geq S_d + 3$$

has no solution (as might occur when precedence and resource constraints interact) can potentially take time proportional to the difference between the upper and lower bounds on the variables. This means that solving scheduling problems with resource contention using bounds propagation can be inefficient, particularly if good bounds are not known. This is discussed further in Section 5, where we present some computational results for scheduling problems.

The utility of UTVPI constraints is not limited to examples like the above scheduling problem, where each constraint clearly falls into the UTVPI category. If groundness information is present, one can deduce that constraints that might appear to lie outside the class actually fall within it. Consider the following implementation of `fib/2`, for (naïvely) computing Fibonacci numbers:

```
fib(N, F) :- N = 0, F = 1.
fib(N, F) :- N = 1, F = 1.
fib(N, F) :- N >= 2,
             N1 = N - 2,
             fib(N1, F1),
             F = F1 + F2,
             N2 = N - 1,
             fib(N2, F2).
```

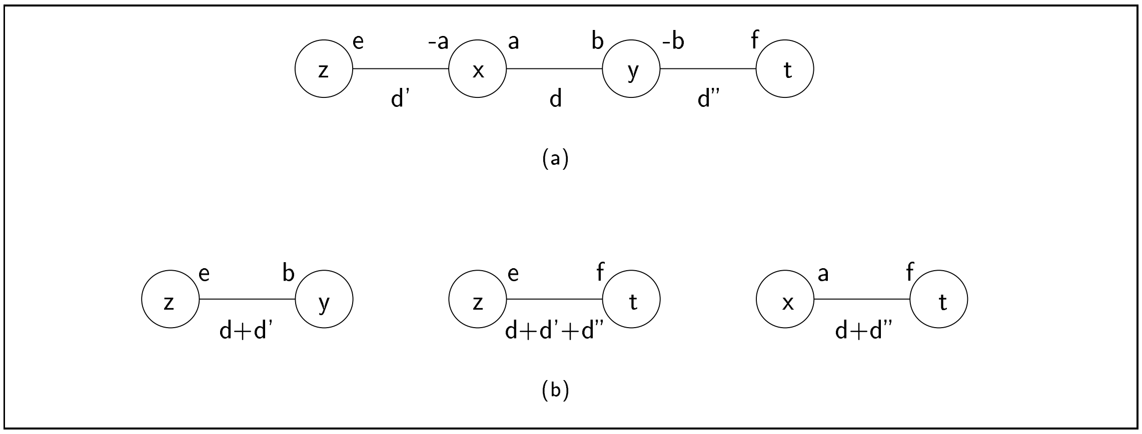

Figure 2: Transitive closure - general case

Although the equation `F = F1 + F2` does not fall into the class of unit two-variable-per-equality (UTVPE) constraints, we know that at the point at which it is reached in the computation, `F1` is fixed to some value g (this information can be generated automatically by program analysis). As a result, the equation is essentially `F = g + F2`, which is UTVPE, and hence can be handled automatically at run-time by a UTVPI solver (handling of UTVPE constraints is discussed in more detail later).

3.1 A UTVPI Constraint Solver

The UTVPI (and SVPI) constraint solver is based on a transitive closure algorithm, which generates all the UTVPI consequences of the input UTVPI constraints. This allows incrementality because, given we have already determined the transitive closure of the previous UTVPI constraints, maintaining the transitive closure is a local process, only involving constraints that share variables with the new UTVPI constraint. An SVPI constraint $ax \leq d$, $a \in \{-1, 1\}$ can be thought of as a UTVPI constraint involving a dummy variable v_0, that is, $ax + 0v_0 \leq d$, and treated analogously.

Note that transitive closure of UTVPI and SVPI constraints yields only UTVPI and SVPI constraints, but that the generated SVPI constraints may not have unit coefficients. For example, combining $x + y \leq 4$ with $x - y \leq -1$ yields $2x \leq 3$. Clearly, this can be replaced by the equivalent (for integers) constraint $x \leq 1$, and in general $\pm 2x \leq d$ may be replaced by $\pm x \leq \lfloor \frac{d}{2} \rfloor$. This means that we can maintain unit coefficients throughout, but more importantly, by incorporating this *tightening* process into the transitive closure algorithm we make the method complete for integer variables. Jaffar et al. [4] show the following theorem:

Theorem 1 *[4] Let C be a set of UTVPI constraints that is closed under transitivity and tightening. The C is satisfiable iff it does not contain a constraint of the form $0 \leq d$ where $d < 0$.* □

The algorithm of [4] alternately transitively closes and tightens a set of UTVPI constraints. In contrast our algorithm requires only one joint step.

Figure 2 illustrates the general case for computing transitive closure. Each node represents a variable, and each arc an inequality. We also allow a node labelled v_0 (as above), which represents any variable which appears with a coefficient of zero, allowing us to

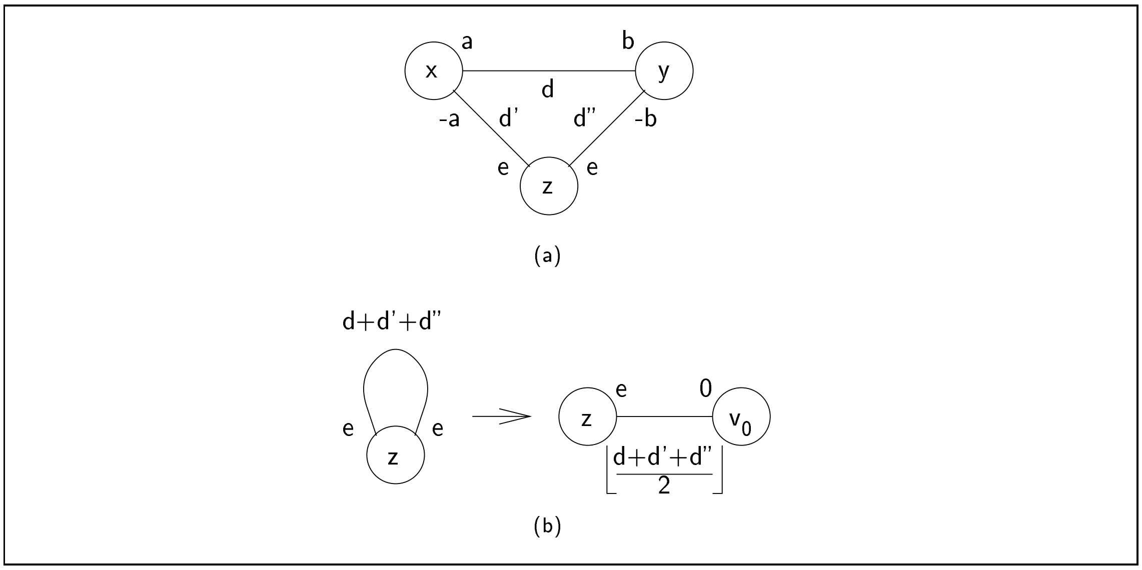

Figure 3: Transitive closure - one special case

represent SVPI constraints. The graph in part (a) represents the inequalities

$$\begin{aligned} ez - ax &\leq d' && \text{(1a)} \\ ax + by &\leq d && \text{(1b)} \\ -by + ft &\leq d'' && \text{(1c)} \end{aligned}$$

Suppose (1a) and (1c) were existing inequalities, and (1b) is a new one. Then by eliminating x and/or y, we obtain the transitive implications

$$\begin{aligned} ez + by &\leq d + d' \\ ez + ft &\leq d + d' + d'' \\ ax + ft &\leq d + d'' \end{aligned}$$

as shown in part (b). This is the most general case when computing the new transitive closure. Special cases arise when more than one of the nodes in Figure 2 corresponds to the same variable, or when one or more are labelled v_0. For instance, Figure 3 shows the case where $t \equiv z$ and $f = e$. The transitive implication is $ez + ez \leq d + d' + d''$, which after tightening becomes

$$ez \leq \lfloor \frac{d + d' + d''}{2} \rfloor$$

We now describe the algorithm, which is essentially just transitive closure with tightening. Rather than use the dummy variable v_0 we treat SVPI constraints directly.

We maintain two sets of constraints. Let T be the set of UTVPI constraints, and S the set of SVPI constraints. After adding each new inequality, as well as testing for satisfiability, we update S and T to ensure that $S \cup T$ remains transitively closed, with all constraints in S having been tightened if necessary. Initially, S and T are both empty.

Suppose we are adding a new UTVPI constraint $c \equiv ax + by \leq d$ $(x \neq y)$. We assume that $S \cup T$ is (integer) feasible.

The first thing we do is check for infeasibility. Adding c results in infeasibility exactly when at least one of the following inequalities does not hold:

$$
\begin{array}{llr}
\{0 \leq d + d' & : \ -ax - by \leq d' \in T\} & \text{(2a)} \\
\cup\ \{0 \leq d + d' + d'' & : \ -ax \leq d' \in S, -by \leq d'' \in S\} & \text{(2b)}
\end{array}
$$

If we have infeasibility, then there is no more to do (the solver returns *false*). Otherwise, we proceed to update S and T. The new set of UTVPI constraints, exactly corresponding to Figure 2, is

$$
\begin{array}{llr}
T' = T \cup \{c\} \cup \{by + ez \leq d + d' & : \ -ax + ez \leq d' \in T, z \neq y\} & \text{(3a)} \\
\qquad \cup\ \{ax + ft \leq d + d'' & : \ -by + ft \leq d'' \in T, t \neq x\} & \text{(3b)} \\
\qquad \cup\ \{ez + ft \leq d + d' + d'' & : \ -ax + ez \leq d' \in T, z \neq y, & \\
 & \quad -by + ft \leq d'' \in T, t \neq x, t \neq z\} & \text{(3c)}
\end{array}
$$

The new set of SVPI constraints (essentially just a set of special cases of the general transitive closure) is

$$
\begin{array}{llr}
S' = S \cup \{by \leq d + d' & : \ -ax \leq d' \in S\} & \text{(4a)} \\
\quad \cup\ \{ax \leq d + d' & : \ -by \leq d' \in S\} & \text{(4b)} \\
\quad \cup\ \{ez \leq d + d' + d'' & : \ -ax \leq d' \in S, -by + ez \leq d'' \in T, z \neq x\} & \text{(4c)} \\
\quad \cup\ \{ez \leq d + d' + d'' & : \ -by \leq d' \in S, -ax + ez \leq d'' \in T, z \neq y\} & \text{(4d)} \\
\quad \cup\ \{by \leq \left\lfloor \frac{d+d'}{2} \right\rfloor & : \ -ax + by \leq d' \in T\} & \text{(4e)} \\
\quad \cup\ \{ax \leq \left\lfloor \frac{d+d'}{2} \right\rfloor & : \ ax - by \leq d' \in T\} & \text{(4f)} \\
\quad \cup\ \{ez \leq \left\lfloor \frac{d+d'+d''}{2} \right\rfloor & : \ -ax + ez \leq d' \in T, -by + ez \leq d'' \in T\} & \text{(4g)}
\end{array}
$$

We illustrate this with an example. Suppose the existing constraints are

$$S = \{-x \leq -2, z \leq -3\}, \quad T = \{x + z \leq 1, y + z \leq 3\}$$

and the new constraint being added is $x - y \leq -2$ (i.e. $a = 1, b = -1$). We first check for infeasibility: in this case the set of constraints to check is empty. We then compute the new set of UTVPI constraints (relevant rules marked):

$$T' = T \cup \{x - y \leq -2\} \cup \{x + z \leq 1 \ (3b)\}$$

and the new set of SVPI constraints:

$$S' = S \cup \{-y \leq -4 \ (4a), z \leq -1 \ (4c)\}$$

If the constraint to be added is not TVPI, but rather an SVPI constraint $c \equiv ax \leq d$, then feasibility is maintained if the following condition holds:

$$\{0 \leq d + d' \ : \ -ax \leq d' \in S\}$$

The set of UTVPI constraints remains unchanged (i.e. $T' = T$), but updating the set of SVPI constraints is as follows:

$$S' = S \cup \{c\} \cup \{by \leq d + d' \ : \ -ax + by \leq d' \in T\}$$

Theorem 2 *The algorithm computes the tightened transitive closure of* $\{c\} \cup S \cup T$.

Proof: Straightforwardly, the algorithm produces only tightened constraints. It is easy to show that the algorithm computes sound transitive consequences. It remains to show that the transitive consequences are complete. This is simple through exhaustive case analysis, except for one case. Suppose during the algorithm we produce an SVPI constraint which needs to be tightened (i.e. it was produced by one of (4e)–(4g), with the sum of the ds being odd). It is not clear why this information (the tightening) need not be re-propagated. Assume that propagating this SVPI constraint through some UTVPI constraint will produce a new SVPI constraint which is not redundant after the algorithm has completed. We prove that this leads to a contradiction for case (4e) — the other cases follow similarly.

For rule (4e), the constraint being tightened is $2by \leq d + d'$ with $d + d'$ odd, having been generated by combining the new constraint being added to the store $ax + by \leq d$ (5) with an existing constraint from the store $-ax + by \leq d'$ (6). We write $d + d' = 2k + 1$, so the tightened constraint is $by \leq k$. Now suppose that propagating this tightened constraint through a UTVPI constraint yields an irredundant constraint $ez \leq k'$ (7). Then the UTVPI constraint that was propagated through to produce this constraint is $-by + ez \leq k' - k$ (8). Note that this constraint must have already been in the constraint store (since we don't generate any constraints involving $-by$), as was (6), and so their transitive closure must have been as well: $-ax + ez \leq d' + k' - k$ (9). However, rule (4g) ensures that during the transitive closure process we combine (5), (8) and (9), to produce

$$ez \leq \lfloor \frac{d + d' + 2(k' - k)}{2} \rfloor = \lfloor \frac{2k + 1 + 2(k' - k)}{2} \rfloor = k'$$

Thus (7), which we assumed was irredundant, is not, because it is also generated by a direct application of (4g). □

The formulation above ignores the issue of redundancy. For each pair of variables x, y, there are only four possible combinations of unit coefficients, so there are only four possible irredundant UTVPI constraints. Similarly, for any variable, there are a maximum of two irredundant SVPI constraints. This means we need store no more than $2n^2$ inequalities for any system of n variables. Adding a new constraint generates no more than $O(n^2)$ transitive consequences ($O(n)$ for SVPI constraints), thus giving us an upper bound of $O(mn^2)$ on the complexity of adding m constraints involving n distinct variables.

4 Beyond UTVPI constraints

UTVPI constraints are able to express some problems of interest, but many problems can not be modelled using just UTVPI constraints. Obviously we are interested in extending the solver to handle larger classes of constraints. We begin by extending the solver to equations and disequations. Then we define a solver for arbitrary integer constraints, using the complete UTVPI solver as the core.

Equality Constraints

We can straightforwardly extend the solver to handle UTVPE constraints, by substituting an equivalent pair of UTVPI constraints. For example, $t - y = 2$ can be replaced by the equivalent $t - y \leq 2 \ \wedge \ -t + y \leq -2$. However, it is more efficient to treat these specially using Gaussian elimination. We can write the equation in the form $var = term$ (in the above case, it is $t = y + 2$), and use this to replace later occurrences of t. We denote the set of such equalities E. Substitution within a UTVPI (or UTVPE) constraint using a UTVPE is guaranteed to give a UTVPI (resp. UTVPE) result, although we may require tightening (for example substituting $y+2$ for t in $t+y \leq 3$ results in $2y \leq 5$ or equivalently $y \leq 2$).

Suppose we are continuing our example from the previous section, but have added an equation, so the constraint store looks like this:

$$\begin{aligned} E &= \{t = y + 2\} \\ S &= \{-x \leq -2, z \leq -1, -y \leq -4\} \\ T &= \{x + z \leq 1, y + z \leq 3, x - y \leq -2\} \end{aligned}$$

Let the new UTVPI constraint to be added be $-t - z \leq -5$. The first thing we do with the new constraint is replace the occurrence of t, so it becomes $-y - z \leq -3$. Having done this, we check satisfiability and add it to the constraint store as before.

Handling equations specially makes it worthwhile detecting *implicit equalities*. For example, the new constraint being added ($-y - z \leq -3$) and the existing constraint $y + z \leq 3$ together imply that $y + z = 3$. We can thus place the equality $y = -z + 3$ (or $z = -y + 3$) in E, and remove all inequalities that include y (resp. z) from S and T (the fact that we are maintaining the transitive closure ensures they are redundant). Note that since we choose y as the variable to eliminate, we also substitute it out of E (a Gaussian elimination step) so that if we ever encounter t again, only one substitution step is necessary. This yields a constraint store that looks like this:

$$\begin{aligned} E &= \{t = -z + 5, y = -z + 3\} \\ S &= \{-x \leq -2, z \leq -1\} \\ T &= \{x + z \leq 1\} \end{aligned}$$

Implicit equalities are detected whenever we add a UTVPI constraint $ax + by \leq d$ to T and T already contains $-ax - by \leq -d$, or S already contains $-ax \leq d'$ and $-by \leq -d - d'$; or when we add $ax \leq d$ to S which already contains $-ax \leq -d$. Removing implicit equalities and performing substitutions decreases the work required by further transitive closures since the number of variables involved in UTVPI constraints is reduced.

Disequality constraints

The simplest additional kind of constraint to consider extending the solver to is unit two-variable-per-disequality (UTVPD) constraints. That is, constraints of the form $ax + by \neq d$ where $a, b \in \{-1, 0, 1\}$. The addition of this class of constraint leads to a satisfiability problem that is NP-complete (they can express graph colouring), so we cannot expect to have a complete solver. But we can handle them incompletely quite straightforwardly using the UTVPE information.

UTVPD constraints are stored, and when equations are discovered (either directly or implicitly) the substitution of the equation is applied to the the UTVPD constraints. If the resulting UTVPD constraint is of the form $0 \neq 0$ (after simplification) then the solver returns *false*, otherwise it returns *unknown*.

An example of why this constraint solver is incomplete can be seen in the following unsatisfiable set of constraints which are transitively closed, and yet not obviously unsatisfiable: $\{0 \leq x, x \leq 1, 0 \leq y, y \leq 1, x - y \neq 0, x - y \neq 1, x - y \neq -1\}$

Arbitrary constraints

In order to solve arbitrary problems involving integer constraints, we need to handle constraints involving more than two variables, and constraints having non-unit coefficients. Propagation based solvers handle all of these constraints, albeit incompletely. Hence we consider augmenting a UTVPI solver with a propagation solver. The simple model of cooperation we outline communicates between the two solvers using SVPI constraints, which both handle completely.

A propagation solver works roughly as follows: each variable is represented by a domain which defines the (possibly infinite) set of values it may take. Associated with each variable is a list of propagation rules which may need to be evaluated when the domain of the variable changes, in order to update the domains of other variables. When a new constraint is added to the system, it creates a number of new propagation rules. These are added to the lists of rules for the variables involved and placed on a queue of rules that are pending evaluation. Propagation rules are removed from the queue one by one. Each rule has a "target" variable, and uses the domain information of the variables in the body of the rule to determine information about the domain of the target variable. If this information about the domain removes some values from the present domain of the variable, then all the propagation rules that may use this information to produce new information are added to the queue. The propagation process continues until unsatisfiability is detected (when some variable's domain becomes empty) or the queue becomes empty.

The propagation solver we built for augmenting the UTVPI solver follows the scheme above, using methods similar to the indexical approach used in clp(FD) (described in [1]). There are two main differences. First we allow unbounded variables. Propagation rules are delayed until the variables involved have sufficient bounds for them to be applicable. Second we allow negative numbers. Excluding negative numbers makes the handling of multiplication and division easier. Presently domains of variables are represented only as upper and lower bounds and a list of disequalities. This approach is inefficient for disequality reasoning when domains of the variables are small, and this behaviour can be seen in the empirical results. We intend to extend the implementation to handle small ranges and gaps in ranges more efficiently.

We now discuss the merger of the two solvers. In the cooperating model, constraints from the derivation are sent to the UTVPI solver if they are UTVPI, and to the propagation solver if they are non-UTVPI. SVPI constraints are sent to both solvers. Each solver deals with its own constraint store. When the UTVPI solver determines a new SVPI constraint then a special entry is placed in the propagation solver's queue of propagation rules which in effect adds that constraint. When the propagation solver determines a new SVPI constraint, it calls the UTVPI solver to add the new SVPI constraint. Either solver can detect failure. Propagation and transitive closure continue until the process terminates.

Note that for the correctness of this communication method, the UTVPI solver that is combined with the propagation solver cannot handle UTVPE (or UTVPD) constraints specially. This is because, while substitution is simple to perform on UTVPI, UTVPE and UTVPD constraints, it is not so simple to adjust the propagation solver to handle substitutions. Hence the simple SVPI communication approach removes some of the advantages of dealing with the unit-two-variable constraints directly, in particular UTVPD constraints cannot be handled directly. There are a number of methods to overcome this which we are currently exploring.

	utvpi	utvpi-noeqn	prop	comb	eclipse
sched-simple-orig	0.004	0.003	0.004	0.004	0.017
sched-simple-slack	0.005	0.004	0.127	0.004	0.297
sched-bridge-first	0.35	0.45	0.23	0.48	0.39
sched-bridge-good	9.01	10.16	5.34	10.70	6.93
job-shop-10x10-std	2.19	2.46	1.02	2.57	1.61
job-shop-10x10-mod	3.25	3.69	4.20	3.95	4.76
job-shop-10x15-std	19.50	21.20	2.96	22.43	5.98
job-shop-10x15-mod	23.27	24.25	11.55	25.34	18.83
fib-utvpe-forw	0.13	0.16	0.44	0.17	2.09
fib-std-forw	—	—	0.55	0.33	2.53
fib-utvpe-back	0.36	0.42	1.22	0.47	5.32
fib-std-back	—	—	1.49	0.93	6.10

Table 1: UTVPI Benchmarks

5 Computational Experience

To evaluate the efficiency of the UTVPI solver we have examined a number of standard integer problems for constraint logic programming. We tested two different versions of the UTVPI solver. The first version, `utvpi`, is the "full" version, as described in this paper, including the handling of disequalities. The other version, `utvpi-noeqn`, has no equation support (and thus no disequation support), to allow the assessment of the cost/benefit of the special handling of equations. In this version, equations are simply treated as a pair of inequalities, without being extracted or stored separately, and disequations are not handled.

We compare our UTPVI solvers against a bounds propagation solver (`prop`) (briefly described in Section 4), together with a combined UTVPI and propagation solver (`comb`), which is essentially the combination of `utvpi-noeqn` and `prop`, as described in Section 4. All these solvers make use of the same underlying run-time engine, a byte code emulated WAM type PROLOG engine based on that of CLP($\mathcal{R}$) [3], hence this gives us the fairest possible comparison of the effect of the solver on the benchmarks. As an independent gauge on the resulting execution we also give times for ECL^{i}PS^e v3.5.1, using the Finite Domains extension library (`eclipse`), which is also based on an interpreted WAM type PROLOG engine. Other widely available finite-domain solvers such as clp(FD) and ILOG SOLVER use quite different run-time approaches.

The benchmarks are written in such a way as to minimise any algorithmic differences between the benchmark code for each system (again in order to try to compare just the solvers). In particular, this means that heuristics such as first-fail have not been used. Since the UTVPI solvers do not have a notion of domain, selecting variables on the size of the domain will give different orders of variable labelling for each system. Hence each system would traverse a different tree, solving a different set of constraints, and a comparison would not be particularly meaningful.

The benchmarks were run on a SPARCstation 10/402. The times shown are in seconds, collected using the built-in timing predicates for each solver, and except where times were over 400 seconds, they are the average of 20 runs.

The first sets of examples only involve constraints which are UTVPI or UTVPE. The first of these are scheduling problems (`sched-*`), with precedence constraints and resource contention, as discussed in Section 3. For each problem we computed worst case bounds a priori (the sum of all task durations) and used these as bounds for all variables.

The first set of scheduling problems (sched-simple-*) use data from the simple scheduling example presented in Section 3, and involve just finding a first solution. sched-simple-orig is the original example, as stated in the text. sched-simple-slack is the same core problem, but with a large extra independent task in parallel. The results for this and the other UTVPI problems are in Table 1. The poor performance of prop and eclipse on these problems is a direct result of an unsatisfiable transitive loop, which takes a bounds propagation solver time proportional to the size of the bounds to discover. The UTVPI solver, on the other hand, discovers the unsatisfiability in time independent of the size of the bounds, since it is a simple UTVPI check (2a). These benchmarks show the dependence of propagation-based solvers on having good bounds.

The other scheduling problems are more realistic. The first (sched-bridge-*) uses data from the bridge construction example in [8]. The tests included finding a first solution, and a "good" solution. The remainder (job-shop-*) are job shop scheduling benchmarks in two sizes: 10×10 (job-shop-10x10-*) and 10×15 (job-shop-10x15-*), looking for first solutions. The first form (std) is the standard form of the problem. While searching for first solutions for this problem, unsatisfiable transitive loops never occur. Hence we modified the problems (-mod) so that each job is made up of 2 tasks on each of 5 different machines (rather than 1 task on 10 machines). In this case the unsatisfiable transitive loop problem does occur.

The systems tested all follow the same search path, and find the same solutions in each case. Thus the UTVPI solver does not detect unsatisfiability where the propagation solvers do not. The only advantage the UTVPI solver then has is the ability to detect unsatisfiable transitive loops faster. In only one case (job-shop-10x10-mod) did this lead to an improvement in execution time. Disappointingly the overhead of maintaining the transitive closure overwhelmed the advantage for detecting unsatisfiable loops faster. As the size of the problem increased this disadvantage worsened. Note, however, how the times for the UTVPI solvers for the std and mod problems are much closer than for the propagation solvers, illustrating some robustness of UTVPI solvers.

The second UTVPI/UTVPE problem set are Fibonacci programs and goals. fib-utvpe-* is the program in the UTVPE formulation presented in this paper, while fib-std-* is the standard (naïve) formulation, with the two recursive calls last. Both versions were run in the forward (in, out) and backward (out, in) directions. *-forw determines the 14th Fibonacci number (610), while *-back determines the position of the Fibonacci number with value 610 (14).

The UTVPI-based solvers win here, with the propagation solvers running significantly slower. The reason for this is that most constraints are used only once for propagation. Hence the set up of the propagation rules is a significant overhead in this problem. Interestingly, the "standard" formulation is slower than the UTVPE formulation on all systems capable of running it. comb is the fastest solver on the "standard" formulation, indicating that despite the fact that it is not a UTVPI program, there are enough UTVPI constraints involved for a UTVPI solver to yield a benefit.

Note that in all the UTVPI benchmarks directly supporting equations in the UTVPI solver provides a noticeable improvement.[2]

We now examine some problems where disequations (UTVPD constraints) are used as well as UTVPI and UTVPE constraints. The first set are temporal reasoning problems, taken from [7], using the "shared-track" knowledge base (temp-st-*). The programs which

[2] For the fib programs it would be much better if we handled new variables in equations specially.

	utvpi	prop	comb	eclipse
temp-st-sfenabled	0.06	0.11	0.06	0.11
temp-st-sfexits	0.29	0.35	0.29	0.34
temp-st-path	0.48	0.75	0.50	0.80
hamil-10	0.02	0.02	0.01	0.02
hamil-20	0.04	87.48	66.33	432.18
hamil-30	1.68	?	?	?
hamil-40	6.56	?	?	?
hamil-50	3530.09	?	?	?
queens-fc-5	0.05	0.05	0.03	0.12
queens-fc-8	3.06	2.02	1.34	4.90
queens-fc-11	321.47	176.22	117.50	398.34
queens-gfc-5	0.04	0.04	0.03	0.03
queens-gfc-8	2.17	1.80	1.63	0.69
queens-gfc-11	259.60	217.33	232.12	59.55

Table 2: UTVPD Benchmarks

originally used disjunction to represent disequations are modified to use disequations directly. The results are given in Table 2. For these benchmarks the UTVPI-based solvers improve upon the bounds propagation ones.

The next benchmark is a program for calculating Hamiltonian paths described in [6] (hamil-*). This problem is representable using only UTVPD, UTVPE and SVPI constraints, and is a best case for the utvpi solver because it handles UTVPD constraints directly. The reason is that unsatisfiability can be detected from UTVPE and UTVPD constraints alone. Because the other solvers do not handle UTVPD directly their search space is much larger. The benchmark was run on five different data sets, ranging in size from 10 nodes to 50 nodes, searching for a first solution. Entries in the table of the form "?" indicate that the benchmark took in excess of 12 hours to run.

The final benchmark in this class in the well-known "N Queens" benchmark. We evaluate two versions: the "forward checking" (queens-fc-*) version, and the more usual "generalised forward checking" (queens-gfc-*) version (see [8] for details), searching for all solutions using board sizes of 5, 8, and 11. The fc version is essentially all SVPD and SVPE constraints, while the gfc version is essentially all UTVPD and SVPE constraints. The results for the fc class show that synergy between the constraint solvers is possible. Here comb is faster than both utvpi and prop. The results for the gfc class show that handling UTVPD constraints directly (as in utvpi) does not benefit unless there are UTVPE constraints also in the problem. Hence prop performs far better than utvpi, and the synergy seen before in comb disappears. For the gfc class, prop (and hence comb) performs badly compared to eclipse principally because of its poorer handling of variables with small domains.

Finally we examine a number of traditional finite domain benchmarks which involve constraints that are not unit coefficient or two variable. The results are in Table 3. magic is a (3x3) magic squares problem, zebra is the well known "Who owns the zebra?" problem, crypt* are cryptarithmetic problems from [8] (crypta is send-more-money). The partit-* problems partition the numbers $1 \ldots N$ into two equal-sized groups whose sum and sum of squares are identical. bramsey is a boolean formulation of the problem of finding a 3-colouring of a complete graph with N vertices such that no monochrome triangles exist. partit and bramsey are example programs that come with the clp(FD) system. perm is the permutation generation example from [8], for which we generated 20 random problems

	prop	comb	eclipse
magic	0.103	0.102	0.170
zebra	0.009	0.009	0.027
crypta	0.011	0.012	0.017
cryptb	0.204	0.242	0.324
partit-8	0.02	0.02	0.07
partit-16	0.35	0.44	2.24
partit-24	12.98	17.08	125.99
partit-32	477.41	630.04	8090.57
bramsey-8	0.21	0.16	0.39
bramsey-10	0.44	0.34	0.84
bramsey-12	0.81	0.61	1.59
bramsey-14	11.33	10.22	9.28
bramsey-16	10148.40	7919.96	6159.53
perm-9	0.19	0.16	0.50
perm-10	0.74	0.63	1.60
perm-11	3.54	3.14	7.60
perm-12	12.78	11.77	23.06

Table 3: Non-UTVPI Benchmarks

(using the same technique as in [8]) and computed the harmonic mean of the resulting times.

For the small benchmarks (magic, zebra and crypta) the use of UTVPI constraints does not show any benefit. For cryptb the UTVPI solver overhead again causes an overall slowdown. Similarly for partit (here the extreme differences between `prop` and `eclipse` result because `eclipse` delays multiplication constraints until one multiplicand is ground, unlike `prop`). However, for the bramsey and perm problems there is some advantage in handling UTVPI constraints better.

Overall the combined constraint solver `comb` improved upon the propagation solver in around half the classes of benchmark. As expected, where the improved strength of the constraint solver reduces the search space (hamil), tremendous performance improvements are possible. Similarly, the more direct handling of UTVPE equations seems advantageous when these are a significant proportion of the constraints encountered (fib). The ability to detect unsatisfiable transitive loops more quickly seems only rarely to be of advantage. Because bounds are generally reasonably tight for most of the benchmarks, the number of iterations of bounds propagation required to find unsatisfiability is usually small. Hence the (small) advantage of the UTVPI solver in this detection is nullified by overhead of maintaining the UTVPI data structures.

6 Conclusions

UTVPI constraints are clearly more expressive than the SVPI constraints that form the basis of bounds propagation techniques. They admit an efficient complete solver which can be used as the basis for a more general solver using propagation methods. The UTVPI solver has distinct overhead over a propagation approach due to its maintenance of more complicated data structures. But for a number of problems the UTVPI constraint solver is able to find failure faster often enough so that an overall benefit in execution times arises. Disappointingly the advantage in finding unsatisfiable transitive loops faster rarely seems to benefit overall.

Combining a propagation solver and UTVPI solver leads to improved performance on some examples, and avoids worst case behaviour of the propagation solver. Hence we argue that UTVPI based solvers are worthwhile including as a component of a general integer constraint solver. Doing so yields another option for a constraint programmer to experiment with, which can be of significant benefit in some instances.

The simple combination of solvers we describe here can be improved upon. We are currently investigating how to make a more closely integrated combined solver, in order to handle UTVPE and UTVPD constraints better. There is also scope for transferring constraints from one solver to the other when enough variables are fixed.

References

[1] Daniel Diaz and Phillipe Codognet. A minimal extension of the WAM for clp(FD). In *Proc. of the 10th International Conference on Logic Programming*, pp. 774–790, 1993.

[2] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, Volume 19 / 20, May 1994.

[3] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco*, pages 128–139, June 1992.

[4] J. Jaffar, M. Maher, P. Stuckey and R. Yap. Beyond finite domains. In Alan Borning (editor), *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA, May 1994.

[5] J. C. Lagarias. The computational complexity of simultaneous Diophantine approximation problems. *SIAM J. Comput.*, Volume 14, Number 1, pp. 196–209, Feb 1985.

[6] J.H.M. Lee, H.F. Leung and H.W. Won. Extending GENET for non-binary CSP's. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence*, pp. 338–343, Nov 1995.

[7] Jonathan S. Ostroff. *Temporal Logic for Real-Time Systems.* Wiley, 1989.

[8] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, MA, 1989.