Linear Equation Solving for Constraint Logic Programming

Jennifer Burg[*] Peter J. Stuckey[†] Jason C.H. Tai[‡] Roland H.C. Yap[†]

Abstract

Linear constraint solving in constraint logic programming requires incremental checks of the satisfiability of a system of equations and inequalities. Experience has shown that Gauss-Jordan elimination and the simplex method are efficient enough to be of practical value in the implementation of CLP languages based on linear arithmetic constraints [1, 2, 4, 6, 7, 8]. However, these algorithms must be modified to accommodate the special demands of CLP execution. First, they must be applied incrementally. Secondly, they must co-exist with backtracking. An added consideration is that constraints containing new variables be brought into the constraint set efficiently. Finally, the recognition of any variable for which a unique value has been determined may be necessary for programs with non-linear constraints. In light of the special nature of the CLP constraint-solving problem, it is difficult to make a clear theoretical argument in favor of one constraint solver over another. Empirical comparisons are in order so that the nature of the typical CLP program can be taken into account. The purpose of this paper is to describe and empirically compare a number of direct linear arithmetic constraint solvers for CLP, focusing on programs which contain only equations, or inequalities which immediately become ground.

1 Algorithms

In this section we define the different Gaussian-based algorithms for linear equation solving that we consider.[1] They all use the same basic steps of *forward elimination* and *back substitution.* The incremental equation-solving problem is defined as follows:

Definition [Incremental equation solving] Given a sequence of equations $e_1, \ldots, e_m$ return the smallest index i such that $e_1 \wedge \cdots \wedge e_i$ is unsatisfiable, or $m+1$ if $e_1 \wedge \cdots \wedge e_m$ is satisfiable. A restriction is that e_{i+1} is not accessible until the satisfiability of $e_1 \wedge \cdots \wedge e_i$ is known.

In this discussion we assume the variables over which the linear constraints are written are $x_1, \ldots, x_n$. We assume there is a total ordering upon

[*]Wake Forest University, Winston-Salem, NC 27109, U.S.A, `burg@mthcsc.wfu.edu`

[†]University of Melbourne, Parkville 3052, Australia, `{pjs,roland}@cs.mu.oz.au`

[‡]Royal Melbourne Institute of Technology, Melbourne 3001, Australia `jtai@yallara.cs.rmit.oz.au`

[1]Other strategies such as iterative or interval based linear equation solving methods are beyond the scope of this paper. See for example [3].

the variables, where $x \prec y$ denotes that variable x precedes y in the ordering. We extend the ordering to include ∞ such that $x_i \prec \infty$ for $1 \leq i \leq n$. The ordering that is used in the system is an important parameter. Usually we will insist that variables are ordered by age, newer variables appearing before older variables.

If t is a linear expression $c_0 + c_1x_1 + \cdots + c_nx_n$ where each $c_i, 0 \leq i \leq n$, is a real number, then let $vars(t) = \{x_i \mid c_i \neq 0\}$. A linear equation e over the variables $x_1, \ldots, x_n$ is of the form $c_0 + c_1x_1 + \cdots + c_nx_n = d_0 + d_1x_1 + \cdots + d_nx_n$. Denote by $lhs(e)$ the expression $c_0 + c_1x_1 + \cdots + c_nx_n$ and $rhs(e)$ the expression $d_0 + d_1x_1 + \cdots + d_nx_n$ A *term form* of the equation e, denoted $term(e)$ is the term $t \equiv (d_0 - c_0) + (d_1 - c_1)x_1 + \cdots + (d_n - c_n)x_n$, where $e \Leftrightarrow 0 = t$.

If e is an equation with a term $t \equiv c_0 + c_1x_1 + \cdots + c_nx_n$ corresponding to e, and $x_k \in vars(t), c_k \neq 0$, we say e may be may be written in a *substitution form* for x_k, denoted $subs(e, x_k)$ as follows.

$$x_k = s_0 + s_1x_1 + \cdots + s_{k-1}x_{k-1} + s_{k+1}x_{k+1} + \cdots + s_nx_n \qquad (1)$$

where $s_i = (-c_i/c_k), 1 \leq i \neq k \leq n$. Call x_k the *subject* of such an equation. This form can be viewed as a substitution for variable x_k that will replace it with a linear expression not involving x_k. Let $e' = subs(e, x_k)$, where $subs(e, x_k)$ is given in equation (1) above. If $t \equiv d_0 + d_1x_1 + \cdots + d_nx_n$ is a linear expression, then e' applied to t as a substitution, denoted $t \bullet e'$, is the linear expression

$$(d_0+d_ks_0)+\cdots+(d_{k-1}+d_ks_{k-1})x_{k-1}+(d_{k+1}+d_ks_{k+1})x_{k+1}+\cdots+(d_n+d_ks_n)x_n$$

For example, if e' is the equation $x_3 = 2 + x_1 - 3x_2 + x_4$ in substitution form for x_3, and t is the linear expression $-2x_1 + x_2 + 2x_3$, then $t \bullet e'$ is the expression $4 - 5x_2 + 2x_4$. We extend the notion of substitution to equations in substitution form as follows: if e is of the form $x_i = t$ then $e \bullet e'$ is the equation $x_i = t \bullet e'$.

Let λ denote the empty sequence, and : represent the concatenation operation for sequences. If O is a sequence of m objects then the length of O, denoted $|O|$, is m, and we let $O_i, 1 \leq i \leq m$, represent the i^{th} object in the sequence. Thus $O \equiv O_1 : O_2 : \ldots : O_m$.

1.1 The Generic Solver

We begin by presenting our generic solver, which incrementally applies some type of Gaussian reduction to a sequence of constraints E. The second parameter of generic_solver(E, solver) defines the type of Gaussian reduction to be employed and maintains F in solved form as described below.

1.2 Forward Elimination

A sequence of equations $E = E_1 : E_2 : \cdots : E_m$ is in *forward elimination form* if the following conditions hold:

- Each E_i is in substitution form $x_{r_i} = t_i$.
- For each $1 \leq i < j \leq m$, $x_{r_i} \neq x_{r_j}$ and $\{x_{r_1}, x_{r_2}, \ldots, x_{r_i}\} \cap vars(t_j) = \emptyset$.

E is a sequence of equations, F is a sequence of equations in solved form, and f is an equation in substitution form.

```
generic_solver($E$, solver)
    Let $F = \lambda$
    for $i := 1$ to $|E|$
        $(F, f) :=$ solver($F$, $term(E_i)$)
        if $f \equiv false$ return $i$
        elseif $f \neq trivial$
            $F := F : f$
    return $|E| + 1$
```

The following sequence is in forward elimination form

$$\begin{array}{lcrrrrrr} x_5 & = & 2 & +x_1 & +3x_2 & -x_3 & -x_4 & +2x_6 \\ x_1 & = & -1 & & -2x_2 & +x_3 & & -x_6 \\ x_3 & = & 2 & & +x_2 & & +2x_4 & \end{array} \qquad (2)$$

Our first variant of the solver is given below. generic_solver(E, ffe) defines a solver which employs an incremental variant of Gaussian elimination. Note that the variable to be chosen as the subject of a new equation after forward elimination is given by the function $choose(t)$, which is left unspecified. (A good heuristic is to choose the newest variable.)

F is a sequence of equations in forward elimination form, f is an equation in substitution form, t is a linear expression, v is a variable, and c is a coefficient.

```
ffe($F$, $t$)
    $t :=$ forward_elimination($F$, $t$)
    $f :=$ make_substitution($t$)
    return $(F, f)$
forward_elimination($F$, $t$)
    for $j := 1$ to $|F|$
        $t := t \bullet F_j$
    return $t$
```

```
make_substitution($t$)
    if $t \equiv c$
        if $c \equiv 0$ return $trivial$
        else return $false$
    else
        $v := choose(t)$
        $f := subs(0 = t, v)$
    return $f$
```

This incremental variant of Gaussian elimination maintains the sequence of equations F in forward elimination form, which we define as a *solved form*. In doing so, it uncovers a conflict in the constraint set if one exists. It differs from the standard Gaussian algorithm in that back substitution is not performed.

The ffe solver has the advantage that it makes support for backtracking trivial. Because the solved form F is modified simply by appending a new equation onto the end of the sequence, earlier states of the solved form can be recovered simply by deleting equations from the end.

1.3 Partial Forward Elimination

One of the special characteristics of equational constraints handled by CLP systems is that new variables, that is, variables that have never previously occurred in a constraint, appear quite often. Forward elimination as defined above does not take advantage of the fact that an equation with a new variable does not require any substitutions, since the new variable can immediately be made the subject of the equation.

We can generalize the idea of not applying forward elimination in the case of equations involving new variables, to the idea of applying only part of the forward elimination step to each new equation. That is, we apply forward elimination only until a variable that is a *true parameter* is discovered in the equation. A true parameter (of a term) is defined as a variable which is not the subject of any equation in F and would remain in the term after the substitutions defined by F have been exhaustively applied.

Partial forward elimination makes use of the ordering on the variables, choosing $least(t)$ as the subject of the new equation. If t is a linear expression, then we define $least(t) = x$ where $x \in vars(t) \cup \{\infty\}$ and $\forall y \in vars(t), \quad x \prec y \vee x = y$. $least(e)$ where e is an equation is defined as $least(term(e))$. Note that the relation $\prec$ respects the fact that variables are ordered in terms of age; that is, new variables precede old variables. When new variables appear in a new equation E_i, one of them must be $least(E_i)$, and hence $least(E_i)$ can immediately be made the subject of the new equation without substitutions. Full forward elimination can be implemented efficiently, using a routine similar to that for partial forward elimination, when choosing the least variable as the subject.

A sequence of equations $E = E_1 : E_2 : \cdots : E_m$ is in *pfe form* if the following conditions hold:

- Each E_i is in substitution form $x_{r_i} = t_i$ and $x_{r_i} \prec least(t_i)$.
- For each $1 \leq i < j \leq |E|$, $x_{r_i} \neq x_{r_j}$

The following sequence is in pfe form where $x_i \prec x_j$ iff $i < j$.

$$\begin{array}{rclcccccc} x_2 & = & 2 & & -x_3 & +x_4 & & & \\ x_3 & = & -1 & & & & & +x_6 & -x_7 \\ x_1 & = & 2 & +x_2 & & & +x_5 & & \end{array} \tag{3}$$

generic_solver(E, pfe) maintains F in pfe form, which is also defined as a solved form.

1.4 Back Substitution

One of the extra requirements made of an equation solver in the CLP context is to provide explicit information about the *fixed variables*, that is, those variables which must take a fixed value. This information is used in delay-based activities such as non-linear equation-solving or delayed evaluation of predicates, in clause indexing, and in input/output and other extra-logical operations. While forward elimination suffices for checking the satisfiability

F is a sequence of equations in pfe form, f is an equation in substitution form, t is a linear expression, v is a variable, and c is a coefficient.

```
pfe($F$, $t$)
    $t$ := partial_elimination($F$, $t$)
    $f$ := make_substitution($t$)
    return ($F$, $f$)

partial_elimination($F$, $t$)
    $done := false$
    while ($vars(t) \neq \emptyset$ and
            $done \neq true$) do
        $v := least(t)$
        if $v \equiv lhs(f)$ for some $f \in F$
            $t := t \bullet f$
        else $done := true$
    return $t$
```

of the constraint set, it does not necessarily provide information about fixed variables. Consider the sequence of equations

$$x_1 + x_2 + x_3 + x_4 = 5 : x_2 + x_3 + x_4 = 3 \tag{4}$$

The forward elimination form of the equations is

$$\begin{array}{lcllll} x_1 & = & 5 & -x_2 & -x_3 & -x_4 \\ x_2 & = & 3 & & -x_3 & -x_4 \end{array}$$

From this it is not explicit that x_1 must take the fixed value 2.

One way to discover this information is to *back substitute*, applying the substitutions defined by equations later in the sequence to equations earlier in the sequence. In doing so, we put the sequence of equations E into *parametric form*, defined as follows:

- Each E_i is in substitution form $x_{r_i} = t_i$.
- For each $1 \leq i < j \leq |E|$, $x_{r_i} \neq x_{r_j}$.
- For each $1 \leq j \leq |E|$, $\{x_{r_1}, x_{r_2}, \ldots, x_{r_{|E|}}\} \cap vars(t_j) = \emptyset$.

Equations in parametric form partition the variables into two sets: the variables appearing on the left hand side of exactly one equation (*non-parameters*) and the variables appearing on the right hand side of equations (*parameters*).

For the system (4), after applying the substitution for x_2 to the equation for x_1 we obtain the system in parametric form:

$$\begin{array}{lclll} x_1 & = & 2 & & \\ x_2 & = & 3 & -x_3 & -x_4 \end{array}$$

Now it is clear that x_1 must take the value 2. It is easy to show that if equations E in parametric form imply that some variable x_j must take a fixed value c, then $x_j = c$ appears in E.

generic_solver(E, ffe-fbs) defines an incremental Gauss-Jordan solver. Gauss-Jordan elimination combines forward elimination with back substitution to maintain a parametric form, our third solved form for the equations. This is essentially the algorithm used by both CLP($\mathcal{R}$) [8] and CHIP [4].

Notice that the algorithm applies full forward elimination rather than partial because back substitution will perform the substitutions in any case.

F is a sequence of equations in parametric form, f is an equation in substitution form, and t is a linear expression.

```
ffe-fbs($F$, $t$)
    $t$ := forward_elimination($F$, $t$)
    $f$ := make_substitution($t$)
    if $f \not\equiv false$ and $f \not\equiv trivial$
        $F$ := back_substitute($F$, $f$)
    return ($F$, $f$)

back_substitute($F$, $f$)
    for $j = 1$ to $|F|$
        $F_j := F_j \bullet f$
    return $F$
```

The chief disadvantage of back substitution is the overhead to backtracking. Because we must be able to recapture earlier states of the solver, changes must either be trailed (e.g. [8, 4]), or recovered by reversing the substitution operations (e.g. [6]). In particular, since column pointers are also maintained to ensure that we don't need to search for where back substitution should be applied, this information must also be recovered.

1.5 Partial Back Substitution

Rather than performing full back substitution, it is possible for a CLP equation solver to do only partial back substitution. By this method, the solver may perform fewer operations while identifying most, or even all, of the fixed variables. In the simplest of these schemes, back substitution is done if a fixed variable happens to be recognized. When all the variables on the right hand side (rhs) of an equation for x are known to be fixed, then the equation can be replaced by a simple equation of the form $x = c$. This may allow other variables to be fixed. If F is a sequence of equations, let $groundable(F)$ be the set of equations of the form $x = t$ in F where t is not a constant and for each $y \in vars(t)$ there is an equation $y = c$ in F.

This value-back-substitution approach is used in the solver executed by generic_solver(E, ffe-vbs). ffe-vbs corresponds most closely to an incremental version of the normal Gaussian elimination for linear equations.

Partial back substitution clearly does not uncover all the fixed variables. For example, it would not find any fixed variables in system (4).

Beringer and De Backer propose a novel equation-solving algorithm that, without performing full back substitution, is guaranteed to find all fixed variables. (The idea is mentioned only briefly in [1]. We develop it in more detail here.) The key intuition behind this scheme is based upon maintaining in each equation at least one *true parameter*, that is at least one variable which is not the subject of any other equation. If for an equation $x = t$, $least(t)$ is a true parameter, then x is not fixed. Thus, to identify all fixed variables, we ensure that the least variable on the right hand side of each equation is a true parameter. If this is not possible for some equation F_i, then the subject of F_i must be fixed.

We define a sequence of equations E to be in *BDB form* if

•Each E_i is in substitution form $x_{r_i} = t_i$, and $x_{r_i} \prec least(t_i)$.

F is a sequence of equations in forward elimination form, f is an equation in substitution form, t is a linear expression, x is a variable, and c is a coefficient.

```
ffe-vbs($F$, $t$)
        $t$ := forward_elimination($F$ , $t$)
        $f$ := make_substitution($t$)
        if $f \equiv false$ or $f \equiv trivial$ or $f$ is not of the form $x = c$
            return ($F$, $f$)
        else
            while $\exists F_i \in groundable(F : f)$
                $t := rhs(F_i)$; $x := lhs(F_i)$
                $t$ := forward_elimination($F$ , $t$)
                $F_i := (x = t)$
            return ($F$, $f$)
```

•For each $1 \leq i < j \leq |E|$, $x_{r_i} \neq x_{r_j}$.

•For each $1 \leq i \leq |E|$, $least(t_i) \notin \{x_{r_1}, \ldots, x_{r_{|E|}}\}$.

Assuming $x_i \prec x_j$ iff $i < j$, then the following sequence of equations is in BDB form.

$$\begin{array}{lcllllll} x_1 & = & -1 & +x_3 & -2x_4 & -3x_5 & +x_6 \\ x_2 & = & & -x_3 & & +x_5 & -x_6 \\ x_6 & = & 3 & & & & \\ x_4 & = & & & & 2x_5 & +x_6 \end{array} \tag{5}$$

Below we describe a number of possible solvers for maintaining a system of equations in this BDB form. The new equation is placed in BDB form as follows: First, either partial elimination is applied twice or forward elimination applied so that the two least variables are true parameters. The least variable is chosen to be the subject of the equation. However, the subject of the new equation may be the least variable, and thus the designated true parameter, of some earlier equations in the sequence. Hence, for each of these equations a new true parameter must be found. Either partial elimination or forward elimination is used to find a new true parameter. We examine three of the four possible BDB solvers, BDB-pfe-pbs, BDB-ffe-pbs, BDB-ffe-fbs.

Adding the equation $x_1 = -1$ to the equations (5) using the solvers BDB-pfe-pbs, BDB-ffe-pbs and BDB-ffe-fbs respectively yields the systems:

$$\begin{array}{lcllll} x_1 & = & -1 & & \\ x_2 & = & & -6x_5 & -2x_6 \\ x_6 & = & 3 & & \\ x_4 & = & & 2x_5 & +x_6 \\ x_3 & = & & 7x_5 & +x_6 \end{array} \qquad \begin{array}{lcllll} x_1 & = & -1 & & \\ x_2 & = & -3 & -6x_5 & -x_6 \\ x_6 & = & 3 & & \\ x_4 & = & & 2x_5 & +x_6 \\ x_3 & = & 3 & +7x_5 & \end{array}$$

$$\begin{array}{lcllll} x_1 & = & -1 & & \\ x_2 & = & -6 & -6x_5 & \\ x_6 & = & 3 & & \\ x_4 & = & & 2x_5 & +x_6 \\ x_3 & = & 3 & +7x_5 & \end{array}$$

F is a sequence of equations in BDB form, f is an equation in substitution form, s and t are linear expressions, x and y are variables, and c is a coefficient. The specific algorithmic variant is parameterized by *forw* and *back*.

```
BDB-$forw$-$back(F, t)$
    if $forw$ == ffe then $t$ := forward_elimination($F$, $t$)
    else $t$ := partial_elimination($F$, $t$)
    $f$ := make_substitution($t$)
    if $f \equiv false$ or $f \equiv trivial$ return $(F, f)$
    $t := rhs(f)$; $x := lhs(f)$
    if $forw$ == pfe then $t$ := partial_elimination($F$, $t$)
    for $i := 1$ to $|F|$
        if $x \equiv least(rhs(F_i))$
            $y := lhs(F_i)$, $s := rhs(F_i)$
            if $back$ == fbs then $s$ := forward_elimination($F : f$, $s$)
            else $s$ := partial_elimination($F : f$, $s$)
            $F_i := (y = s)$
    return $(F, f)$
```

Because they perform fewer back substitutions and thus alter fewer rows, the BDB solvers hopefully have less backtracking overhead. Also only a column pointer for the first variable on the right hand side of each equation is required to efficiently implement the back substitution. As a result, in many cases, fewer column pointers may be needed with BDB solvers compared with the ffe-fbs solver.

2 Extensions

An important consideration for equation solvers inside a constraint logic programming system is the accuracy of the solver. Floating point representation of numbers can lead to incorrect answers when the constraints define a numerically unstable system. There are two pivoting methods for improving the accuracy of equation solving: complete pivoting and partial pivoting. Complete pivoting is not possible for incremental problems since the entire problem is not known, but partial column pivoting is implementable by defining *choose*(t) to pick the variable with the largest absolute value of the coefficient. Unfortunately partial pivoting is incompatible with those solvers that use a fixed variable ordering, e.g. pfe, and the BDB solvers. In order to use partial pivoting with ffe the forward elimination algorithm must be implemented with the inefficient method described above, rather than with a more efficient method relying on a fixed variable ordering and doing the substitution steps in the fixed order. Hence partial pivoting is easiest to add to the ffe-fbs solver. We define the solver stable to be ffe-fbs with the partial pivoting variable choice.

Another consideration is the optimization of access dead variables. Typical constraint programs involve intermediate variables which are used for building constraints but then are never referred to in later computation. Maintaining relationships about these no longer used variables, the so called *dead variables*, is unnecessary and wastes solver execution time and space. Global analysis methods [9] are able to determine when a variable becomes *dead*, and the solver can take advantage of this information to simplify the constraint store.

Consider the `sum 1.0` program below. The variable $S1$ is used in two equations, and after the second is never referred to again. (See [9].) Hence it can be removed after the first reference. An optimized version `sum 2.0` adds $dead(V)$ annotations that instruct the solver to remove the variable V.

```
SUM 1.0                     SUM 2.0
sumlist([], 0).             sumlist([], 0).
sumlist(X.Xs, S) :-         sumlist(X.Xs, S) :-
      S = X + S1,                 S = X + S1,
      sumlist(Xs, S1).            sumlist1(Xs, S1).
                            sumlist1([], S) :- S = 0, dead(S).
                            sumlist1(X.Xs, S) :-
                                  S = X + S1, dead(S),
                                  sumlist1(Xs, S1).
```

If a variable is dead and the subject of an equation, then it should be back-substituted out of the remaining equations and the equation involving it should be removed. For ffe-fbs the back substitution will already have been done and thus just the row needs to be removed. Unfortunately, column pointers are required for efficient back substitution. This places an extra overhead on all the solvers other than ffe-fbs, which already has column pointers. The solver dead is the ffe-fbs solver with additional machinery to remove dead variable rows.

3 Empirical Results

In this section we compare our various solver algorithms in terms of execution speed, space, and accuracy. These comparisons were obtained using an experimental version of CLP($\mathcal{R}$).

First we introduce our suite of example programs and goals. The first set of programs are simple: `sum` sums a list of terms, where goal `g1` adds a list of 100 variables, `g2` adds a list of 100 of the same variable, and `g3` adds a list of the form $[1, 2, \ldots, 100]$. `fib` calculates Fibonacci numbers naively, where `g1` calculates the 16^{th} Fibonacci number, and `g2` finds which Fibonacci equals 610; `mort` is a mortgage program, where all goals study a 30 year loan at 12%, `g1` asks for payment given a principle of \$20000, `g2` asks for the principle given the payment, and `g3` asks for relationships between payment, principle and balance. `laplace` determines temperatures on a square plate with 100C on three sides and 0C on one side using a 13 * 13 finite element matrix. `inv` is a matrix inversion program which inverts a 12*12 matrix.

`gaus` tests satisfiability of a system of 100 equations in 100 variables. `ode` is a differential equation-solver (described in [5]) running a large deterministic goal. `msprimes` is a magic square program where all numbers are different primes. `chem` is a large program determining equilibrium constants for chemical reactions. `chess` solves a chess puzzle from Sam Lloyd. `circ` solves a circuit element preference problem for a 16 resistor circuit. `ladder` constructs and solves a ladder resistor circuit of size 100. `mech` reasons about mechanical designs. Finally `gaus-n` attempts to find a satisfiable combination of constraints over 100 variables. There are 25 choices to be made, each between two possible sets of 4 equations, to select a total of 100 equations.

The solver statistics of each program and goal above are given in Table 1. These statistics are intended to be representative of the amount of computation required and are mainly solver independent, except where the program behaves differently because of non-linear constraints and fixed variables. Statistics include the size of the program in number of lines; the total number of equations, variables, inequalities and non-trivial dead variables encountered in the execution of the goal; and the peak number of equations and variables in the solver during execution. Some of the programs do contain linear inequalities, but the amount is either relatively small or the inequalities can be handled trivially. The last column gives the percentage of equations that are encountered with a new variable. (The presence of a new variable often simplifies equation solving). Programs `chem`, `circ` and `mech` contain non-linear constraints which remain non-linear when encountered at execution time, and as such they cannot run correctly on the solvers which do not detect fixed variables. The ffe-vbs solver is able to run `chem`, but not the other two since it does not detect all fixed variables. The stable solver is unable to run `ode` in a reasonable time because back substitution becomes very expensive with the large system of equations in `ode`.

Deterministic goals, that is those that run without backtracking, are annotated (D) or (I) in Table 1. Note that even for deterministic goals, backtracking information must be stored for the solvers that modify rows in the equation solver, since during execution it is not clear that backtracking may not occur. The deterministic goals for which indexing can determine that no backtracking is required are annotated (I).

The solvers we investigate are those defined in the previous section. In all cases except stable we assume $choose(t)$ selects the $least(t)$, and the variable ordering is defined by placing newest variables first. Table 2 gives the total execution times for the benchmarks using each of the solvers normalized to the least such time. In order to be able to differentiate more reliably among the timings, which can be quite close, we have used timings obtained from a software simulator. These results ignore caching effects and pipelining, but certainly agree with median clock times. The last lines are harmonic means for the goals which every solver can run, for deterministic goals only, and for non-deterministic goals for the full solvers.

Empirical results show that overall there is no best equation solver for

Program	Lines	Total				Peak		New%
		Eqs	Vars	Ineqs	Dead	Eqs	Vars	
chem	370	246297	53926	482	—	8153	2520	19
chess	184	217406	7903	5518	—	139394	115	2.2
circ	50	13873	102	0	27	2560	102	0.31
fib(g1) **D**	9	4557	3946	986	984	4004	3946	65
fib(g2)		13298	13161	4057	4000	4004	3946	69
gaus-n	563	2875	102	0	—	460	102	≃0
gaus **I**	268	200	100	0	—	158	100	≃0
inv **I**	35	4464	4032	0	1584	4090	4032	49
ladder **D**	108	567	498	19	355	556	498	33
laplace **D**	32	227	169	0	—	227	169	22
mech	600	16309	12833	643	—	1120	500	52
mort(g1) **D**	8	1141	1083	359	716	1141	1083	67
mort(g2) **D**		1141	1083	359	716	1141	1083	66
mort(g3) **D**		1139	1083	359	716	1139	1083	66
msprimes	19	550403	32	0	—	69	32	≃0
ode **I**	151	81737	78904	0	64602	78964	78904	13
sum(g1) **I**	23	361	303	100	—	361	303	67
sum(g2) **I**		262	203	100	99	262	203	87
sum(g3) **I**		261	203	100	99	261	203	88

Table 1: Statistics for the benchmarks

				ffe-fbs		BDB		
Program	ffe	pfe	ffevbs	ffefbs	stable	pfepbs	ffepbs	ffefbs
chem	——	——	**1.00**	1.64	1.93	1.36	1.44	1.56
chess	**1.00**	1.04	1.40	1.28	1.29	1.23	1.23	1.26
circ	——	——	——	1.15	1.13	1.40	**1.00**	1.09
fib(g1) **D**	1.21	**1.00**	1.55	2.20	2.24	1.46	1.60	1.62
fib(g2)	**1.00**	1.60	1.23	1.56	1.58	1.16	1.27	1.28
gaus-n	2.43	2.43	2.67	**1.00**	1.02	1.62	1.62	1.75
gaus **I**	1.01	**1.00**	1.02	1.10	1.14	1.17	1.18	1.39
inv **I**	1.23	**1.00**	1.42	1.90	2.75	3.77	2.00	2.28
ladder **D**	**1.00**	1.01	1.28	1.57	14.31	1.58	1.53	1.59
laplace **D**	**1.00**	1.21	1.13	1.83	10.59	2.26	1.69	1.93
mech	——	——	——	1.04	1.04	**1.00**	**1.00**	1.01
mort(g1) **D**	1.04	**1.00**	1.27	1.21	28.81	1.33	1.33	1.35
mort(g2) **D**	1.07	**1.00**	1.31	1.29	22.71	1.37	1.38	1.40
mort(g3) **D**	1.24	**1.00**	1.32	1.17	42.21	1.34	1.32	1.32
msprimes	8.66	8.96	1.11	1.04	1.07	**1.00**	1.01	1.04
ode **I**	**1.00**	1.08	1.15	1.59	——	2.24	1.66	1.84
sum(g1) **I**	1.04	**1.00**	1.28	1.37	1.38	1.33	1.34	1.37
sum(g2) **I**	1.05	**1.00**	1.21	1.21	1.22	1.36	1.38	1.43
sum(g3) **I**	1.04	**1.00**	1.20	1.12	1.12	1.22	1.21	1.24
HMean All	1.18	**1.16**	1.30	1.32	1.94	1.40	1.37	1.43
HMean Det	1.07	**1.02**	1.25	1.39	——	1.53	1.44	1.52
HMean NDet	——	——	——	1.20	1.23	1.21	**1.18**	1.24

Table 2: Normalized execution times

use in a constraint logic programming system. However, we can observe that certain solvers are better for certain types of programs. For the programs without non-linears, the ffe and pfe algorithms do well, with two exceptions: `gaus-n` and `msprimes`. In particular, `msprimes` is a bad case for both because they lose the ability to exploit clause indexing since ffe and pfe do not determine all the fixed variables. Both ffe and pfe see effectively 550403 equations, while each of the other solvers sees only 18287 total equations in the execution of `msprimes`, with the remainder handled by clause indexing. The other exception is `gaus-n`. The back substitution phase of ffe-fbs, which simplifies the equation systems in the constraint store, leads to a reduction of work during the exploration of the search tree.

Overall for deterministic goals (D or I) the strategy of doing the least possible work for determining satisfiability (pfe) is the most advantageous. Thus pfe is generally the best choice where the identification of fixed variables is not important. (Unfortunately for real programs fixed variables are usually important.) pfe is always the fastest solver when the goal is deterministic and the percentage of new variables is high ($\geq 50\%$).

When computation is non-deterministic then consideration must be given to speculative work. Thus, the back-substituting solvers fare better, since they simplify the form of constraints to speed up later constraint solving. Of the back-substituting solvers, the BDB-ffe algorithms perform uniformly no more back substitutions than ffe-fbs, and always use less space. They are not, however, generally faster, and are certainly less efficient than ffe-fbs when there is deep backtracking over complex constraints. If we restrict consideration to the large "real" programs `chem, chess, circ, mech` and `ode` then BDB-ffe-pbs becomes the clear winner. Surprisingly it is the "middle path" BDB solver that wins out, avoiding the worst case behavior of BDB-pfe-pbs, and performing uniformly faster than BDB-ffe-fbs.

Some preliminary results using two other prototype experimental CLP systems with linear arithmetic (CLP(Real) and XPI) on a subset of the programs show similar results for the choice of fastest solver. For deterministic programs, ffe and pfe are fastest. Where there is a large amount of non-determinism, then the back-substituting solvers start to gain. CLP(Real) chooses not to use column pointers for the back-substituting solvers, saving space at the expense of requiring time to search for variables to be substituted out of equations. The preliminary results show that the absence of column pointers is crucial to the efficiency of the back-substituting solvers. For example, ffe-fbs becomes about 4 times slower than ffe for `fib g2`. More work is in progress with CLP(Real) and XPI.

Table 3 gives the normalized peak space usage for the solvers. These experiments show that ffe-fbs and stable solvers are clearly the worst in terms of space usage. This is because of the frequent back substitutions, which require that a significant amount of backtracking information be stored. The best solvers in terms of space are generally the ones with no back substitution, ffe and pfe, unless the program is indexed deterministic (I), in which case the solver state does not have to be saved for backtracking. In this case, the BDB solvers are generally more space efficient (because back substitution simplifies the store). The exceptions to this generalization are `inv` and `sum` for goals `g2` and `g3`, which incur significant implementation overhead for maintaining column pointers.

				ffe-fbs		BDB		
Program	ffe	pfe	ffevbs	ffefbs	stable	pfepbs	ffepbs	ffefbs
`chem`	——	——	1.09	1.43	1.39	**1.00**	1.17	1.17
`chess`	1.01	**1.00**	1.06	1.10	1.07	1.07	1.07	1.07
`circ`	——	——	——	1.03	1.07	1.30	1.04	**1.00**
`fib(g1)` **D**	1.86	**1.00**	2.04	2.15	2.15	1.27	2.04	2.04
`fib(g2)`	1.61	**1.00**	1.77	1.77	1.77	1.14	1.77	1.77
`gaus-n`	**1.00**	**1.00**	1.02	8.35	8.35	8.28	8.28	8.28
`gaus` **I**	1.91	1.91	1.94	1.91	1.91	**1.00**	**1.00**	**1.00**
`inv` **I**	1.42	**1.00**	1.59	1.41	1.49	3.09	1.07	1.07
`ladder` **D**	**1.00**	1.03	1.25	1.73	12.39	1.54	1.44	1.44
`laplace` **D**	1.62	**1.00**	1.76	3.49	11.33	3.57	2.55	2.55
`mech`	——	——	——	1.04	1.04	**1.00**	**1.00**	**1.00**
`mort(g1)` **D**	**1.00**	1.25	1.25	1.50	1.75	1.25	1.25	1.25
`mort(g2)` **D**	**1.00**	1.12	1.25	1.50	1.75	1.25	1.25	1.25
`mort(g3)` **D**	**1.00**	**1.00**	1.20	1.40	2.00	1.40	1.20	1.20
`msprimes`	1.16	**1.00**	1.22	1.67	1.59	1.31	1.31	1.31
`ode` **I**	1.70	**1.00**	1.95	2.26	——	2.42	1.52	1.52
`sum(g1)` **I**	1.12	1.12	1.37	1.12	1.12	**1.00**	**1.00**	**1.00**
`sum(g2)` **I**	**1.00**	**1.00**	1.16	1.33	1.33	1.16	1.16	1.16
`sum(g3)` **I**	**1.00**	**1.00**	1.20	1.20	1.20	1.20	1.20	1.20
HMean All	1.18	**1.07**	1.34	1.61	1.83	1.40	1.36	1.36
HMean Det	1.21	**1.09**	1.43	1.59	——	1.42	1.29	1.29
HMean NDet	——	——	——	1.45	1.44	**1.28**	1.35	1.34

Table 3: Peak space usage

Surprisingly in these benchmarks, ffe-vbs captures almost all the fixed variable information that the other back-substituting solvers do (with the exception of `circ`), albeit usually at a later stage. This is because if every variable in the solver is fixed then ffe-vbs will determine it. The advantage of this can be seen for the program `laplace`, where ffe-vbs determines all the fixed variables with much less space than any other back-substituting solver. The disadvantage is shown for `gaus-n`, where the execution time is the worst of all the solvers.

The penalties for a stable solver are clear: increased execution time and space usage (and sometimes vastly increased execution time even to a point where goals will not execute). The advantage can be seen in the graph in Figure 1, which shows accuracy at the center of the matrix as a result of running the `laplace` program on a matrix of size n. An error of k indicates the value is within 10^k of the correct answer. Once the error increases beyond 2, the result is effectively meaningless. All BDB solvers have the same accuracy and are shown by one plot. Dead variable elimination does not change accuracy and is not displayed. The **stable** solver is clearly superior, while the BDB solvers could be considered marginally more stable than the remaining solvers.

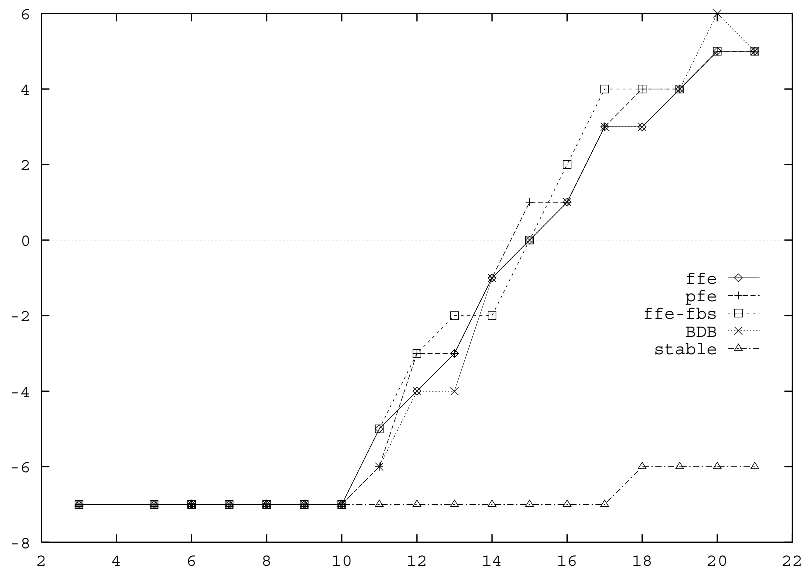

Figure 1: Error 10^k of solvers on `laplace` versus size n

The advantages of dead variable elimination are shown in Table 4. Programs were analysed by hand to determine where (some of) the dead variables occur, and modified versions with dead annotations were constructed. The solver **dead** is run on the annotated versions. The dead variables in the benchmarks are removed by using the dead annotation, which does not make use of the more efficient *in-situ dead removal* techniques. Hence the results can possibly be improved using more sophisticated dead removal. The **dead** solver is compared against the best non-back-substituting solver, **pfe**, and the best back-substituting solver, **BDB-ffe-pbs**. Dead variable elimination drastically improves space usage when it is available, making the **dead** solver the most space efficient. It also usually reduces the execution time of the **ffe-fbs** solver below that of the **BDB-ffe-pbs**. An interesting experiment would be to add dead variable elimination to each of these solvers and compare the effect. The extra overhead of handling full back substitution would indicate that the advantages are not so great as for **ffe-fbs**. This is clearly an area for future work.

	Execution time				Peak solver space			
Program	pfe	ffefbs	BDB	dead	pfe	ffefbs	BDB	dead
`circ`	——	1.45	1.26	**1.00**	——	1.41	1.42	**1.00**
`fib(g1)`	**1.00**	2.20	1.60	1.95	**1.00**	2.15	2.04	1.30
`fib(g2)`	1.26	1.23	**1.00**	1.15	**1.00**	1.77	1.77	1.09
`inv`	**1.00**	1.90	2.00	1.63	1.20	1.70	1.28	**1.00**
`ladder`	**1.00**	1.55	1.52	1.46	**1.00**	1.68	1.40	1.01
`mort(g1)`	**1.00**	1.21	1.33	1.02	2.48	2.97	2.48	**1.00**
`mort(g2)`	**1.00**	1.29	1.38	1.08	2.23	2.96	2.47	**1.00**
`mort(g3)`	**1.00**	1.17	1.32	1.26	2.46	3.45	2.96	**1.00**
`ode`	**1.00**	1.47	1.54	1.13	2.26	5.11	3.44	**1.00**
`sum(g2)`	**1.00**	1.21	1.38	1.09	1.94	2.57	2.25	**1.00**
`sum(g3)`	**1.00**	1.12	1.21	1.12	1.64	1.96	1.96	**1.00**

Table 4: Dead variable elimination

References

[1] H. Beringer and B. De Backer. Combinatorial Problem Solving in Constraint Logic Programming with Cooperating Solvers. In *Logic Programming: Formal Methods and Practical Applications*, C. Beierle and L. Plümer (eds.). Elsevier Science Publishers B. V. 1994.

[2] J. Burg. Parallel Execution Models and Algorithms for Constraint Logic Programming over a Real-Number Domain. PhD Dissertation. University of Central Florida. 1992.

[3] C.K. Chiu, J.H.M. Lee. Interval Linear Constraint Solving using the Preconditioned Interval Gauss-Seidel Method. *Proc. International Conference on Logic Programming.* (This volume). 1995.

[4] M. Dincbas, P. Van Hentenryck, H. Simonis and A. Aggoun. The Constraint Logic Programming Language CHIP. *Proc. of the 2^{nd} Intl. Conf. on Fifth Generation Computer Systems.* 249–264. 1988.

[5] J.A. Harland and S. Michaylov. Implementing an ODE Solver: a CLP Approach. Technical Report 87/92. Department of Computer Science. Monash University. 1987.

[6] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\mathcal{R}_{Lin}$). *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation.* 349–360. 1994.

[7] J-L. Imbert, J. Cohen and M.D. Weeger. An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-Interpreter for CLP. *Journal of Logic Programming.* **16(3-4)**. 235–253. 1993.

[8] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Trans. on Programming Languages.* **14(3)**. 339–395. 1992.

[9] A.D. Macdonald, P.J. Stuckey, and R.H.C. Yap. Redundancy of Variables in CLP($\mathcal{R}$). *Procs. International Logic Programming Symposium.* 75–93. MIT Press. 1993.