Algorithmic Power from Declarative Use of Redundant Constraints

M.H. van Emden
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, B.C., V8W 3P6 Canada

Abstract

Interval constraints can be used to solve problems in numerical analysis. In this paper we show that one can improve the performance of such an interval constraint program by the declarative use of constraints that are redundant in the sense of not needed to define the problem. The first example shows that computation of an unstable recurrence relation can be improved. The second example concerns a solver of nonlinear equations. It shows that, by adding as redundant constraints instances of Taylor's theorem, one can obtain convergence that appears to be quadratic.

1 Introduction

In certain forms of declarative programming, such as logic programming, one is careful to include only just enough clauses as are necessary to define the relations to be queried. Violation of this rule is often punished by multiple identical answers or even by an otherwise terminating program becoming nonterminating[1].

Constraint programming is different. Experience shows that adding more constraints than necessary to define the problem at worst makes finding solutions somewhat slower, and often makes it considerably faster or more accurate. This is the phenomenon that we study in this paper. In the remainder of this introduction we outline what type of constraint processing we are concerned with and what type of application.

There are two approaches to constraint processing. In both forms, the solutions to the constraint system are the n-tuples that satisfy all constraints. In the first approach one attempts to build up an individual n-tuple by using the constraints to fill in single values for the as yet unassigned variables. The second approach maintains at all times for each variable the set of values that have not been shown inconsistent. In this approach the associated sets always contain all solution values. In this paper we are concerned with this latter approach, the *consistency method*.

In the consistency method, computation consists of *constraint propagation*, which considers in turn each of the constraints and removes values that are inconsistent with the constraint. Propagation halts when none of the sets changes. In this stable state, the remaining sets sometimes identify the solution, or a sufficiently small set containing a solution. If the remaining candidate sets are too large, then disjoint constraint systems are spawned with more restricted value sets. This disjunction process can be iterated as much as required to yield sufficient accuracy in the form of sufficiently small sets of possible values. In discrete applications "sufficiently small" usually means being a singleton set. In real-valued applications it usually means an interval of a width in the same order of magnitude as the desired error in numerical analysis would be.

The consistency method has been used with great success in operations research and in combinatorial problems more generally [32, 16, 13]. Following Cleary's pioneering paper [10], the BNR Prolog team [6, 23, 24] showed that the consistency method also has important advantages in solving numerical problems. (That is, in solving constraint problems where the unknowns are reals.) A theoretical foundation was established in a report dated 1993, which was recently published [5]. BNR Prolog is now available as ALS Prolog. Other implementations are Prolog IV [4], and Numerica [14]. Independently of this, and preceding it, there is a long tradition in Russia of "subdefinite calculation" of which Unicalc [28] is a result.

2 Redundant constraints

Redundant clauses may not just slow down a logic program, but often prevent it from terminating. The worst that redundant constraints do seems to be a mild slowing down. However, considerable improvements in accuracy or in speed can result.

When the constraints are just enough to specify the so-

[1] There is an interesting exception: the addition of previously computed results as redundant clauses. These can replace multistep deduction by a direct look-up. This technique is known as "memoization" [17] or "tabling" [29, 7, 27, 8].

lution, it is often the case that a great deal of search is necessary to obtain a solution. The phenomenon is that if one adds redundant constraints to a minimal set sufficient to specify the solutions, much, or sometimes even all, search is avoided and a great savings in computation results.

The method of redundant constraints has been exploited with great effect in problems with discrete-valued variables [13, 2]. An especially powerful example is Zhou's use [33] of redundant constraints to obtain what seems to have been, at the time at least, the most efficient program for solving job-shop scheduling problems. And this is a purely declarative approach. Speed-ups in scheduling applications by means of redundant constraints are also reported in [9].

In spite of such successes of redundant constraints in the discrete consistency method, it seems that they have not been used in the consistency method with real-valued variables. In this paper we show that redundant constraints can also have worthwhile effects in this situation.

We present two applications, both to well-known problems in numerical analysis. The first concerns recurrence relations, where instability can annihilate all information about the solution. The second concerns solving a nonlinear equation, where we propose a constraint processing counterpart of Newton's method.

3 What are interval constraints?

In this section, we give a brief introduction to interval constraints, beginning with an example.

Let us consider the problem of computing the x and y coordinates of one of the intersections of a circle and a parabola. Thus we consider the interval constraint system:

$$x^2 + y^2 = 1 \wedge y = x^2 \wedge 0 \leq x \qquad (1)$$

This is a conjunction of three logical formulas related by sharing x and y, which denote unknown reals.

The logical formulas are regarded as constraints on the possible values of the unknowns. An interval constraint implementation contracts intervals by removing values that are inconsistent with the given constraints. As a result all solutions, if any, are contained within the computed intervals.

Complex constraints such as those occurring in the left-hand side of the first formula of interval constraint system (1) cannot be processed directly by efficient methods. Instead, the interval constraint system given in formula (1) is translated to primitive constraints, as shown in formula (2). The translation process introduces the auxiliary variables x_2 and y_2:

$$x^2 = x_2 \wedge y^2 = y_2 \wedge x_2 + y_2 = 1 \wedge y^2 = x \wedge 0 \leq x \qquad (2)$$

$x + y = z$
$x * y = z$
$x^n = y$ for integer n
$x = y$
$x \leq y$

Figure 1: Primitive Constraints.

The most commonly used primitive constraints are shown in the table in Figure 1.

As each of the variables is regarded as an unknown real, it is associated with an interval containing all values of this real that occur in any solution. Rather than "solving" such a system, one removes values from the intervals that are inconsistent with the constraints. Of the various degrees of consistency, it has been found that "local consistency" is usually a feasible way of getting the intervals close the limits of what the computer's floating-point number system can attain.

With each type of primitive constraint there is a *constraint contraction operator* that contracts the given intervals for the unknowns occurring in the constraint by removing inconsistent values. Iterating the contraction operator to achieve local consistency is called *constraint propagation*. For more information about constraint contraction operators and constraint propagation, see [5, 6, 31, 14]. Here we only show an example of the operator and give a brief sketch of an algorithm.

An example of constraint contraction Consider, for example, constraints of the form $u + v = w$, one of which occurs in the example. Suppose that the intervals for u, v and w are $[0, 2], [0, 2]$ and $[3, 5]$ respectively. Then all three intervals contain inconsistent values. For example, $v \leq 2$ and $w \geq 3$ imply that $u = w - v \geq 1$. Hence the values less than 1 for u are *inconsistent*, in the conventional sense of the word: it is inconsistent to believe the negation of the conclusion of a logical implication if one accepts the premise. Similar considerations rule out values less than 1 for v and values greater than 4 for w. Removing all inconsistent values from the *a priori* given intervals leaves the intervals $[1, 2]$ for u and v and $[3, 4]$ for w. The new bounds 1 and 4 are computed according to the rules of interval arithmetic and require the rounding direction to be set appropriately. Thus the method of interval constraints depends on interval arithmetic [1, 12, 20].

A sketch of a propagation algorithm An interval constraint system performs an interval contraction for each of the primitive constraints. In the untypical case that constraints do not share variables, the contraction operator has to be applied only once. Any application in a situa-

tion where the intervals have not changed since the last application has no effect. Because of this, the contraction operator is deactivated after being applied.

However, constraints typically share variables. As a result, contraction has to be performed multiple times on any given constraint: every time another constraint causes the interval for a variable to contract, all constraints containing that variable have to be activated again. Because changes are always contractions and because interval bounds are floating-point numbers, a finite number of contractions suffices to reach a state where all constraints yield a null contraction. The constraint propagation algorithm terminates when this is found to be the case.

There are several variants of propagation algorithms. See [18, 16, 5, 19].

As the consistency method only removes inconsistent values and may not remove all such values, it may be that the resulting intervals contain no solution. As a consequence, results of constraint propagation have the meaning: *if* a solution exists, then it is in the intervals found.

A numerical example For example, let us solve the system (1) with BNR Prolog, starting with $[0, 1]$ for x and y. In this case the initial interval happens to be a stable point for the propagation algorithm. A single bisection step is sufficient in this case to let propagation proceed to machine precision. Actually, it suffices for the lower bound for x to be increased from 0 to 0.0000001 to obtain as intervals for x and y respectively $[0.786151377757422, 0.786151377757425]$ and $[0.618033988749893, 0.618033988749897]$.

4 Unstable recurrences

In one of the first quantum-mechanical investigations of the water molecule, Coolidge [11] needed to evaluate $c(n) = \int_0^1 x^n e^x dx$ for $n = 1, 2, \ldots$ Integration by parts yields the recurrence relation $c(n+1) = e - (n+1) * c(n)$. Applying this to the known value of $c(0) = e - 1$ gives, in a typical combination of compiler and processor, a sequence such as that in Figure 2. In this figure we only show correct decimals up to $c(18)$. From there onwards, the numerals are marked with an asterisk to indicate that no decimals are correct. This is easy to see, as $0 \leq c(n) \leq 1$ for $n > 0$. The incorrect decimals are shown to give an indication of the rapidly increasing error.

When we realize that $c(n)$ is monotone decreasing and that $0 < c(n) < 1$ from $n = 2$ onwards, it is clear that the accuracy deteriorates rapidly: the recurrence is unstable.

As an interval constraint system cannot tell a lie (in the sense that only inconsistent values are removed from the

```
c( 0) ~ 1.71828182845905
c( 1) ~ 1.0 (exact)
c( 2) ~ 0.718281828459045
c( 3) ~ 0.56343634308191
c( 4) ~ 0.464536456131406
c( 5) ~ 0.3955995478020
c( 6) ~ 0.344684541647
c( 7) ~ 0.305490036930
c( 8) ~ 0.274361533016
c( 9) ~ 0.24902803132
c(10) ~ 0.2280015153
c(11) ~ 0.21026516
c(12) ~ 0.1950999
c(13) ~ 0.181983
c(14) ~ 0.17052
c(15) ~ 0.1604
c(16) ~ 0.15
c(17) ~ 0
c(18) ~ -0.2   (*)
c(19) ~ 6.6    (*)
c(20) ~ -129.3 (*)
```

Figure 2: Result of typical combination of compiler and processor to compute $c(0) = e-1$, $c(n+1) = e-(n+1)*c(n)$ without intervals.

candidate intervals), it is interesting to use this method to obtain information about the values of these integrals.

We have used BNR Prolog [22] to run the examples of this paper. For those not familiar with Prolog, first a few words about the syntax of this language. Function definitions are organized around the symbol `:-`. To the left of `:-` is the function heading; to the right is the function body. Function bodies consist of function calls interspersed with constraints. The latter are distinguished by being surrounded by braces. Functions may have multiple definitions. They get tried in the order as written. Execution of a function body consists of performing the function calls as defined or adding the constraints to the constraint store. Thus Prolog acts like a programming front end to a constraint solver. It allows one to generate parametrized constraints according to possibly recursive definitions.

We have used BNR Prolog to set up an interval constraint system consisting of $c(0) = e - 1$ and $c(n+1) = e-(n+1)*c(n)$ for $n = 0, \ldots, 19$. This was done with the following program:

```
//E is Euler's constant 2.718...
euler(E)
:- X: {real(2.718281828459045,
            2.718281828459046), X = E}.

sequence(N,N,X,Z,[X|Z]).
sequence(K,N,C1,L,Z)
:- K < N, N1 is K+1,
   {C2: real}, euler(E),
   {C2 = E - N1*C1},
   sequence(N1,N,C2,[C1|L],Z).

result(W)
:- euler(E), {X: real, X = E-1},
   sequence(0,20,X,[],Z), reverse(Z,W).
```

Each instance of $c(n+1) = e - (n+1) * c(n)$ is translated to the constraint `C2 = E - N1*C1` and added to the constraint store. The recursively defined predicate `sequence` creates a list containing $c(n)$ for $n = 0, \ldots, 20$. This list is, after some formatting, shown in Figure 3. Intervals are given in condensed notation. For example, `0.5634363430819 [02,16]` stands for `[0.563436343081902,0.563436343081916]`.

The above inclusions hold in spite of the rounding errors made in computing the bounds. The instability of the recurrence relation shows by the intervals becoming too wide to be of use. That is, we have a true statement about, say, c(20), but it gives no useful information.

In constraint programming, when confronted with such a situation, one should consider adding a redundant constraint. For example, from the definition $c(n) = \int_0^1 x^n e^x dx$

```
c( 0) = 1.7182818284590 [4,5]
c( 1) = [0.999999999999999,1.0]
c( 2) = 0.71828182845904 [3,8]
c( 3) = 0.5634363430819 [02,16]
c( 4) = 0.464536456131 [381,439]
c( 5) = 0.39559954780 [1852,2141]
c( 6) = 0.344684541646 [198,7936]
c( 7) = 0.3054900369 [23494,35662]
c( 8) = 0.27436153 [2973746,3071092]
c( 9) = 0.24902803 [0819221,1695329]
c(10) = 0.2280015 [11505755,20266836]
c(11) = 0.210265 [105523847,201895746]
c(12) = 0.195 [099405710093,10056217288]
c(13) = 0.1819 [74520211601,89554227843]
c(14) = 0.170 [428069269244,638545496629]
c(15) = 0.1 [58703646009609,61860789420381]
c(16) = 0.1 [28509197732957,79023492305296]
c(17) = [-0.32511754073098,0.533625466998779]
c(18) = [-6.88697657751898,8.57039756161668]
c(19) = [-160.119271842258,133.57083680132]
c(20) = [-2668.69845419793,3205.10371867362]
```

Figure 3: Result of propagation from constraints obtained from the unstable recurrence. No redundant constraint used.

```
c( 0) = 1.7182818284590 [4,5]
c( 1) = 1.0 []
c( 2) = 0.71828182845904 [5,6]
c( 3) = 0.5634363430819 [09,10]
c( 4) = 0.46453645613140 [7,8]
c( 5) = 0.3955995478020 [09,10]
c( 6) = 0.34468454164698 [7,8]
c( 7) = 0.3054900369301 [28,34]
c( 8) = 0.27436153301 [7974,8019]
c( 9) = 0.24902803129 [6873,7279]
c(10) = 0.2280015154 [86257,90312]
c(11) = 0.210265158 [065618,110217]
c(12) = 0.195099931 [136445,671634]
c(13) = 0.1819827 [16727802,23685253]
c(14) = 0.170523 [696865499,794269819]
c(15) = 0.16042 [4914411766,6375476561]
c(16) = 0.1514 [59820834074,83197870783]
c(17) = 0.143 [067464655739,464874279783]
c(18) = 0.1 [35914091422952,43067464655739]
c(19) = 0. [0,135914091422952]
c(20) = [0.0,2.71828182845905]
```

Figure 4: Result of propagation from constraints obtained from the unstable recurrence. The only difference with Figure 3 is the addition of the redundant constraint $0 \leq c(n) \leq 3$.

---

it is clear that, say, $0 \leq c(n) \leq 3$. Accordingly, we modify the above program so that the line

```
C2 = E - N1*C1,
```

becomes

```
C2 = E - N1*C1, 0 =< C2, C2 =< 3,
```

No other modifications are made. As a result the output becomes as shown in Figure 4.

The differences between the tables in Figures 3 and 4 are inconspicuous, but important. They occur in $c(2)$ through $c(6)$. For example, the interval for $c(2)$ has improved, even though its width was in Figure 3 already down to about 10^{-15}. And this improvement is caused by adding the constraint $0 \leq c(n) \leq 3$, which specifies an interval larger by *fifteen orders of magnitude*.

In numerical analysis it is well-known how to get accurate results in recurrences such as $c(n+1) = e - n * c(n)$. Here the inevitable rounding error is multiplied by n at each stage, thus increasing exponentially. A remedy is to apply the recurrence in the backward direction, as in $c(n) = (e - c(n+1))/(n+1)$. This backward recurrence could be started at, say $n = 100$. Any error in $c(100)$ rapidly becomes negligible when one applies the backward recurrence.

All this is simple, and widely known. The point is that, with interval constraints, such analysis is not needed: an obvious redundant constraint is sufficient to remedy the problem. This addition (of the redundant constraints `0 =< C2, C2 =< 3`) is the *only* change required. In particular, the forward form of the recurrence, namely `C2 = E - N1*C1`, does not need to be changed to the backward form.

Such a degree of declarativity helps make numerical computation safe for non-analysts. To judge our success at increasing precision, we only need to look at the width of the intervals.

A final note on what happens for systems larger than the one for $c(0), \ldots, c(20)$ discussed so far. It is instructive to set up the same interval constraint system in all respects, except that the constraints are generated for $c(0), \ldots, c(100)$. To distinguish the two systems, we will denote them $c_{20}(0), \ldots, c_{20}(20)$ and $c_{100}(0), \ldots, c_{100}(100)$.

Without the redundant constraints, we found that it takes longer for c_{100} to build up wide intervals: $c_{100}(49)$ is the first that has an interval wider than $c_{20}(20)$.

With the redundant constraints, it takes much longer for wide intervals to appear: $c_{100}(91)$ is the first interval that is as wide as the one for $c_{20}(6)$. Thus, the short sequence c_{20} is misleading. It gives the impression that most of the intervals are wide. c_{100} suggests that only the last dozen or so have wide intervals.

This is to be expected when one sees the analysis that leads conventional computation to choose the backward version of the `C2 = E - N1*C1`. This analysis is also applicable to interval constraint systems, as the translation to primitive constraints combined with constraint contraction gives the effect of both forward and backward computation. The redundant constraint is applied everywhere, therefore also at $c_{100}(100)$. The coupling of the constraints then causes the same effect as the use of a backward recurrence. This would suggest that it is the distance from the last constraint that determines the width of the intervals. According to this suggestion, intervals would start to get larger from $c_{100}(86)$ onwards. In fact, it only happens from $c_{100}(91)$ onwards.

Similarly, c_{100} behaves better without the redundant constraint. We have no explanation for this improvement, except for the general observation that a larger constraint system never has fewer opportunities for constraint contraction.

5 Solving nonlinear equations

As next example of the power of redundant constraints, let us consider finding roots of a function f and compare Newton's method, as used in numerical analysis, with how this problem can be solved effectively and elegantly with redundant constraints.

In interval constraints all one needs to do is to enter the constraint $f(x) = 0$. Constraint propagation gives an interval for x, usually not the smallest, containing all roots. In case there are no roots, a nonempty interval may still be returned: remember that only inconsistent values are removed, and many non-root values for x are not inconsistent. To go from such a large interval to small intervals for the typically existing multiple roots of a nonlinear equation, one splits the interval initially obtained, and spawns separate interval constraint systems for each of the two halves. Such splitting is repeated as far as necessary to obtain intervals of the desired small width.

This process is to be compared to the usual bisection method in numerical analysis (for a generic reference, see [26]). To obtain faster convergence, numerical analysis uses Newton's method [26]. We first derive Newton's method and then show how redundant constraints can yield a version of Newton's method that is an improvement in being both more declarative and more effective.

6 Newton's method

Newton's algorithm for finding a zero of a one-variable function is heuristically justified by Taylor's theorem, which, in its general form, states that for all reals x and x_0, and for all $f : \mathcal{R} \to \mathcal{R}$ that have the required continuous derivatives,

$$f(x) = \sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i + \frac{f^{(n)}(\xi)}{n!}(x - x_0)^n, \quad (3)$$

where ξ is a real between x and x_0. Among hundreds of alternative sources, I just mention my favourite, which is [15].

In the general form of Newton's method, we take for x a root of f, so that $f(x) = 0$. We assume that x_0 is a good approximation to this root, so that $|x - x_0|$ is small. That might suggest discarding the remainder term, so that one gets

$$0 \approx \sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i.$$

Solving this for $x - x_0$ gives an estimate for the root x, because x_0 is known. In general, this solving process is fraught with numerical difficulties, even for n as small as 3. That is why, to obtain Newton's algorithm, one takes $n = 2$, giving

$$0 \approx f(x_0) + (x - x_0)f'(x_0).$$

Solving this for x gives the following formula for improving an estimate x_0 to x_1:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)},$$

which is the usual form of Newton's method.

7 Redundant constraints for finding roots

Let us look at the problem of finding roots with the knowledge that an interval constraint system can be made more effective by the addition of redundant constraints. As in the example of the unstable recurrence, we have a constraint system that completely defines the solution, in this case $f(x) = 0$. The knowledge that Newton's algorithm is to a certain extent effective and is based on the Taylor expansion of f might then suggest that we add as redundant constraint the result of applying a Taylor expansion to the $f(x)$ in $f(x) = 0$, which is the statement of the problem.

An unsatisfactory feature of Newton's method is the necessity to use a truncated form of the Taylor expansion:

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0).$$

Unsatisfactory because either we have the undefined relation $\approx$ or, if we regard this relation as equality, we have a false assertion of which we hope that the consequences are not too serious in the vicinity of the root.

The declarative nature of interval constraints allows us to copy straight from the calculus book [15]:

$$f(x) = f(x_0) + (x - x_0)f'(\xi) \quad (4)$$

where

$$(x_0 \le \xi \le x) \vee (x \le \xi \le x_0). \quad (5)$$

So far, no algorithm. Constraint processing algorithms do arise when we want to automate the construction of the interval constraint system to be solved. Here we have to decide which instances of (4) to include in our interval constraint system. We can look for an advantageous choice of x_0. As we obtain more information about the location of a zero, we will have successively more advantageous values and add redundant constraints accordingly. This suggests the following meta-algorithm for finding solutions of $f(x) = 0$.

Taylor Constraints algorithm
solve $f(x) = 0$ yielding interval X for x with midpoint m;
while width$(X) > \epsilon$ do
    add the constraints (4) and (5) with $x_0 = m$;
    solve the resulting interval constraint system

The disjunction (5) is handled in the usual way: two independent interval constraint systems are created. Such a split occurs every time such an instance of Taylor's theorem is added. Most of the alternative interval constraint systems thus generated are terminated because of inconsistency. The ones that remain are small gaps between zero-free segments of the real line.

Thus we obtain a global algorithm for locating zeros by merely adding instances of a declarative statement of a mathematical theorem. The only algorithmic component of the process is that x_0, the undetermined parameter in the theorem, is chosen to be the midpoint of the current interval for the zero.

A program for Taylor Constraints The above informal description of the Taylor Constraints method can be expressed in a BNR Prolog [22] program as shown in Figure 5.

The disjunction (5) in Taylor's theorem finds a natural translation to :

```
between(X,Xsi,X0) :- {X =< Xsi, Xsi =< X0}.
between(X,Xsi,X0) :- {X >= Xsi, Xsi >= X0}.
```

This translation is a declarative definition of a relation between three reals. In BNR Prolog it has an algorithmic effect. Suppose that `X0` is the middle of the interval for `X`, as it is in the program of Figure 5. The effect is that the existing interval for the zero is bisected and that alternative interval constraint systems are generated for each half. This is exactly what is programmed explicitly in other algorithms for finding roots. Here it happens automatically by the action of a general-purpose inference system acting on a purely declarative statement of a theorem.

Numerical example. In response to the query `?- go(Z).` the program in Figure 5 prints out the following bounds for `Z`:

```
-1.0e+100            1.0e+100
-1.50916975768387    4.18727501261463
-0.838271496392374   1.33905262746538
0.250390565536504    1.33905262746538
0.794721596500944    1.33905262746538
0.794721596500944    1.06688711198316
0.976444490412722    1.02173699672681
0.999995450894563    1.000004590866
0.999999999999997    1.0
```

```
f(X,Y) :- {Y = X**4-4*X**3+4*X**2-4*X+3}.
/* Has roots 1, 3, i and -i. */
f1(X,Y) :- {Y = 4*X**3-12*X**2+8*X-4}.
/* First derivative. */

between(X,Xsi,X0) :- {X =< Xsi, Xsi =< X0}.
between(X,Xsi,X0) :- {X >= Xsi, Xsi >= X0}.

/* taylor1(X): add first-order instance of
   Taylor's theorem.
*/
taylor1(X)
:- X0 is midpoint(X), {Xsi: real},
   between(X,Xsi,X0),
   {Y: real}, f(X0,Y),
   {Y1: real}, f1(Xsi,Y1),
   {0 = Y + (X-X0)*Y1},
   /* After applying Taylor constraint */
   print(X), nl.

eps(0.0000000000001). /* for example */

/* taylor(Zero): if the interval for Zero is
   too large, then add as constraint the
   first-order instance of Taylor's theorem.
*/
taylor(Zero)
:- W is delta(Zero), eps(Eps), W =< Eps, !.
taylor(Zero) :- taylor1(Zero), taylor(Zero).

go(Zero)
:- {Zero: real}, print(Zero), nl,
   f(Zero,0),
   print(Zero), nl, taylor(Zero).

?- go(Zero).
```

Figure 5: A BNR Prolog program for first-order Taylor Constraints.

The second interval is the one resulting from $f(x) = 0$ by itself. The third results from adding a Taylor constraint for the first time. The next three intervals are still so wide that only bisection applies. We can tell, because one of the bounds is still the same as in the previous interval. Only in the last three do both bounds improve and here we see the doubling of the number of correct decimals that is typical of quadratic convergence.

When one asks BNR Prolog for a next answer, it continues with:

```
1.33905262746538  4.09043708443892
1.33905262746538  2.71474485595215
2.71474485595215  3.62336210458554
2.71474485595215  3.16905348026885
2.97523430587216  3.0718221396914
2.99655297044649  3.00253251842071
2.99999901641466  3.00000098679923
3.0               3.0
```

The same quadratic convergence shows.

8 The effect of Taylor constraints as redundant constraints

In our example of an unstable recurrence we presented a controlled experiment demonstrating the effect of redundant constraints: we compared two versions of a program only differing in the presence of the redundant constraint. For the problem of solving an equation, we showed in the previous section that a quadratically convergent sequence of intervals for a root can be obtained by adding suitable instances of Taylor's theorem to the logically minimal definition `f(x,0)` of the zero.

But roots of equations can also be obtained to the same degree of precision by repeatedly applying the definition `f(x,0)` in disjoint intervals, without the use of Taylor constraints. Thus what is still lacking is a controlled experiment as in section 4: results from two versions of the same program, computing a root to the same precision, only differing in the presence of the redundant Taylor constraint. In this section we present such a comparison.

Figure 6 shows a BNR Prolog program that computes the roots of the equation specified by `f(x,0)` to the precision determined by the argument of `eps`. The Taylor constraint is commented out. Therefore, in the version shown, the root is computed without the use of Taylor constraints; only on the basis of the defining constraint $f(x, 0)$, which occurs in the definition of `split`. This we call Version A. When the comment delimiters are removed, the Taylor constraints kick in; that is Version B.

The predicate `splittable(X,M)` (definition omitted) checks whether the interval for `X` is wide enough to be

```
f(X,Y)  :- {Y = X**4 - 12*X**3 + 47*X**2 - 60*X}.
f1(X,Y) :- {Y = 4*X**3 - 36*X**2 + 94*X - 60}.

eps(0.0000000001).

split(X) :- f(X,0), split1(X).

split1(X) :- splittable(X,M),!,split2(X,M).
split1(X).

split2(X,M)
:- forget(count(N)), N1 is N+1,
   remember(count(N1)),
   {X =< M},
/* {[Xsi,FM,F1Xsi]: real, X =< Xsi, Xsi =< M},
   f(M,FM), f1(Xsi,F1Xsi),
   {0 = FM + (X-M)*F1Xsi}, */
   split1(X).
split2(X,M)
:- forget(count(N)), N1 is N+1,
   remember(count(N1)),
   {X  > M},
/* {[Xsi,FM,F1Xsi]: real, M =< Xsi, Xsi =< X},
   f(M,FM), f1(Xsi,F1Xsi),
   {0 = FM + (X-M)*F1Xsi}, */
   split1(X).

q(T,N)
:- new_state(0), new_state(1000),
   T is cputime,
   remember(time(T)), remember(count(0)),
   {X: real}, split(X), fail.
q(T,N)
:- recall(time(T1)), T is cputime-T1,
   recall(count(N)).
```

Figure 6: Program for computing the roots of the equation specified by `f` to the precision determined by the argument of `eps`.

	version A		version B	
epsilon	msec	nodes	msec	nodes
1	60	18	675	18
10^{-1}	270	130	1975	74
10^{-2}	1320	530	2725	94
10^{-3}	2396	912	2501	94
10^{-4}	4021	1418	2755	98
10^{-5}	4911	1806	2766	98
10^{-6}	6252	2190	2651	100
10^{-7}	7867	2700	2581	100
10^{-8}	8446	3088	2911	102
10^{-9}	9532	3476	2816	104
10^{-10}	10832	3988	3045	104

Figure 7: Test results

split, and, if so, makes `M` the midpoint of that interval.

The intended query for this program is

```
?- q(Time,Nodes).
```

The answer will substitute for the variables the number of milliseconds taken and the number of splits effected. The definition of `q(T,N)` accumulates these data by updating extra-logical state variables so that their values are preserved under backtracking. The first clause for `q(T,N)` forces failure in order to suppress output.

The clause

```
split(X) :- f(X,O), split1(X).
```

sets up the constraint `f(X,O)` defining the solutions and then uses `split1(X)` to generate additional constraints forcing `X` to belong to subintervals of the original interval for `X`. In this way the original interval is searched for solutions.

This search is effected by adding to the existing interval constraint system either the constraint `X =< M` or the constraint `X > M`, where `M` is the middle of the original interval for `X`. If either fails, then we have proved that no root is contained in the half thus defined. In case of nonfailure, the same search continues recursively until the interval for `X` is no wider than the argument of the predicate `eps`.

The predicates `forget`, `remember`, and `recall` use BNR Prolog's extra-logical state-recording information. It causes a term `count(N)` to be updated every time a node of the search tree is created.

Discussion of measurements In the table in Figure 7 version A and version B are compared as to the required computing time and number of splits effected (nodes in the binary tree of subintervals). Each line reports one run for each version, with the same accuracy required of both versions. These accuracies run from 1 to 10^{-10}.

The binary tree of subintervals contains $2 * 2^n - 1$ nodes if n is the depth of the tree (we count the depth of a tree consisting of one node as 1). With the smallest epsilon, n is about 30. The number of nodes visited is a very small fraction of the total number nodes in the tree. Interval constraints, augmented or not with Taylor Constraints, prune this tree very effectively.

At low accuracies, there is no scope for quadratic convergence. Hence, at low accuracies, version B has no advantage; it has only the disadvantage of the extra processing required by the Taylor constraint. At higher accuracies, the effect of the redundant Taylor constraint is sufficient to more than offset this extra processing and to net an overall gain in time equal to a factor of about three.

One can summarize these findings by observing that the Taylor constraint achieves a drastic reduction in the number of nodes visited at the expense of an increasing amount of processing per node. Version A has a more or less constant amount of processing per node, and suffers from a linearly increasing number of nodes visited. Version B has a more or less constant number of nodes visited. The pruning is achieved at an increasing cost per node. However, this increase is considerably slower than the increase in the number of nodes visited by Version A.

Note that constraint propagation, even without the Taylor constraint (Version A), is already pretty impressive: its search space grows approximately linearly with the *exponent* in epsilon.

9 Related work

The interval Newton method [20, 21] represents an interesting stage in between the original Newton method and our contribution of using Taylor's theorem as redundant constraint. Interval Newton is an improvement on its predecessor, which merely drops the remainder term in the Taylor expansion. Interval Newton uses a correct version of Taylor's theorem by substituting an interval for the remainder term. However, though this substitution is correct, it discards some information that is present in the original form. The Taylor Constraints method described here may be the first that both achieves correctness and avoids information loss by using the original form of the theorem. For a more detailed comparison, see [30].

Interval constraints, being based on the consistency method, only removes inconsistent values. As a result, one is not assured that a root occurs in the final interval. One only knows that all roots, if any, are contained in the union of the final intervals. Experience shows that if final intervals are small, they almost always do contain a root. A unique feature of Interval Newton is that in many cases it can confirm conclusively that an interval contains at least

one root, or even that it contains exactly one root. The method can also be incorporated into Taylor Constraints; see [30].

A striking feature of Newton's method is its asymptotic quadratic convergence for simple zeros. Interval Newton has this property as well. The examples shown in this paper suggest that Taylor Constraints have the property as well. A proof is beyond the scope of the present paper; but see [30].

Benhamou, McAllester, and Van Hentenryck [3] also use as starting point interval constraint propagation on $f(x) = 0$. They observe that the result of constraint propagation leaves an unnecessarily wide interval for x, and use Newton's method to compute an improvement on each *bound* separately. Hence the quadratic convergence applies to a precise location of the bound. It does not imply that the width of the interval converges quadratically to zero, which is a more useful property. This question does not seem to have been addressed before.

10 Conclusions

Redundant constraints are a new area of investigation in which it is not clear how to proceed. Below we discuss a number of questions.

In what sense can a constraint be redundant? In the example of the unstable recurrence it was clear what the redundant constraint was, because by itself it did not define the solution. Here the redundant constraint cannot be otherwise, because it cannot serve as definition.

However, in the example of the Taylor constraint, either the original constraint by itself or the Taylor constraint by itself can define the solution. We designated the latter as the redundant one only because it is not the simplest.

How should one investigate redundant constraints? Other investigations have the potential of casting light on the effect of redundant constraints, but fail to do so by not comparing the performance of constraint propagation in the two cases: (1) definition only and (2) definition plus redundant constraint.

An example is the work of Zhou [33] on job-shop scheduling. Zhou was interested in two things: being the first to solve certain well-known unsolved complex instances of the job-shop scheduling problem and to investigate several novel techniques for deriving redundant constraints. In addition to this investigation, it would have been interesting to see the effect of *any* redundant constraint added to those that are just enough to define the problem.

Zhou's work is testimony to the power of redundant constraints: they allowed his program to be the world's best job-shop scheduling program, at least for a time. What is needed in addition is *controlled experimentation* comparing the performance of a program with the redundant constraint against the same program without. This is of course not possible if the faster program is barely fast enough to solve the problem in question, as in Zhou's work. For such comparisons one needs some unchallenging problems, as in this paper.

In this paper we have investigated redundant constraints in the time-honoured way of comparing performance between versions with or without the ingredient under investigation. In agriculture and medicine this method is perfectly acceptable. It would be disappointing if a more scientific way cannot be found: in a mathematically based discipline, such as computing, one hopes for theory to help answer such questions. At the moment such theory seems to be lacking.

When can one expect a redundant constraint to help? It would seem that choosing for a redundant constraint some random reformulations of the definition may not help. We do not know whether this has been investigated. It is not an attractive avenue of research. But it would be embarrassing if it did work and is avoided because of lack of intellectual glamour.

Of the known successful examples, most are selected because of their mathematical significance. Taylor's theorem is an example in our work. Zhou [33] uses an important combinatorics theorem of P. Hall on distinct representatives. In the present paper a trivial redundant constraint was also used ($0 \leq \int_0^1 x^n e^x dx \leq 3$). But this was not random: it was chosen to counteract the observed ignorance of this fact.

Do redundant constraints only prune search spaces? In an optimization problem one not only has to satisfy a constraint, but to minimize an objective function as well. In fact, Zhou's work is of this form. There redundant constraints have the clear function of pruning the search space. In our second example, the Taylor constraints indeed pruned the search space considerably, as shown in Figure 7. In the first example, however, there is no search space. It seems that redundant constraints are about more than just pruning search spaces.

11 Acknowledgements

To Bill Older I am indebted more than the references to his work can suggest. He supported me with many discussions and explanations and provided me with numerous unpublished notes. Paul Wormer and Frank Roberts helped with the example of the unstable recursion. Terrance Swift

helped with literature references. Jacques Cohen, Jim Lee, and two anonymous referees pointed out errors and made many suggestions for improvement of the text. Finally, the Natural Science and Engineering Research Council of Canada generously provided research facilities.

References

[1] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations*. Academic Press, 1983.

[2] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20:97–123, 1994.

[3] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proc. 1994 International Symposium*, pages 124–138, 1994.

[4] Frédéric Benhamou, Pascal Bouvier, Alain Colmerauer, Henri Garetta, Bruno Giletta, Jean-Luc Massat, Guy Alain Narboni, Stéphane N'Dong, Robert Pasero, Jean-François Pique, Touraïvane, Michel Van Caneghem, and Eric Vétillard. Le manuel de Prolog IV. Technical report, PrologIA, Parc Technologique de Luminy, Marseille, France, 1996.

[5] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.

[6] BNR. BNR Prolog user guide and reference manual. 1988.

[7] R. Bol and L. Degerstedt. Tabulated resolution for well-founded semantics. In *Proc. of the Symposium on Logic Programming*, 1993.

[8] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[9] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. Speeding up constraint propagation by redundant modelling. In *Lecture Notes in Computer Science*, volume 1118, pages 91–103. Springer-Verlag, 1996.

[10] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987.

[11] Albert Sprague Coolidge. A quantum mechanics treatment of the water molecule. *Physical Review*, 42:189–209, 1932.

[12] Eldon Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker, 1992.

[13] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[14] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.

[15] Einar Hille. *Analysis, volume I*. Blaisdell, 1964.

[16] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[17] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[18] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[19] Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[20] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[21] Arnold Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

[22] William Older. Constraints in BNR Prolog. 1992.

[23] William J. Older. Using interval arithmetic for nonlinear constrained optimization. Bell-Northern Research Technical Report, March 3, 1993.

[24] W.J. Older. The application of relation arithmetic to X-ray diffraction crystallography. Bell-Northern Research Technical Report, February 7, 1989.

[25] W.J. Older, G.M. Swinkels, and M.H. van Emden. Getting to the real problem: Experience with BNR Prolog in OR. In Leon Sterling, editor, *Proceedings of the Third International Conference on the Practical Application of Prolog*, pages 465–478. Alinmead Software Ltd, 1995.

[26] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, Second Edition*. Cambridge University Press, 1992.

[27] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.

[28] Alexander L. Semenov. Solving optimization problems with help of the Unicalc solver. In R. Baker Kearfott and Vladik Kreinovich, editors, *Application of Interval Computations*, pages 211–224. Kluwer Academic Publishers, 1996.

[29] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.

[30] M.H. van Emden. Finding nonzeroes of nonlinear functions. In preparation.

[31] M.H. van Emden. Value constraints in the CLP Scheme. *Constraints*, 2:163–183, 1997.

[32] D. Waltz. Understanding line drawings in scenes with shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[33] Jianyang Zhou. *Calcul de plus petits produits cartésiens d'intervalles*. PhD thesis, Laboratoire d'Informatique de Marseille, 1997.