Bounds Consistency Techniques for Long Linear Constraints

Warwick Harvey and Joachim Schimpf

IC-Parc
Imperial College
Exhibition Road, London SW7 2AZ, UK
wh@icparc.ic.ac.uk
j.schimpf@icparc.ic.ac.uk

Abstract. We present a number of techniques for efficiently achieving bounds consistency for linear constraints with large numbers of variables.

1 Introduction

Bounds consistency is a popular technique in constraint programming over integers and reals [3, 4]. In this paper we examine how to efficiently propagate a linear constraint to achieve and maintain bounds consistency. Our techniques are specifically aimed at constraints with many variables, typically achieving bounds consistency in sub-linear time. We do not address the issue of a more global view of a system of constraints, such as considering multiple constraints simultaneously.

Except where noted, all the techniques apply to either real or integer variables, but see Section 5.4 for a discussion of floating point rounding issues.

This paper is organised as follows. In Section 2 we start by presenting some notation and the bounds consistency algorithm from [1] on which we are improving. In Sections 3 and 4 we present some refinements to the basic propagation technique. In Section 5 we discuss combinations of algorithms, point out some pitfalls and suggest some areas for further investigation.

2 Basic Two-Pass Propagation

Suppose we wish to perform bounds propagation on a constraint

$$\sum_{i=1}^{n} a_i x_i \leq b \tag{1}$$

We assume for simplicity of exposition that the x_i are distinct variables and that $a_i > 0$ for all i. If the x_i are not distinct then the propagation performed is likely to be weaker than necessary, but no more so than with other approaches. If some

of the a_i are negative then some signs, bounds, etc. will be swapped around but otherwise it is the same.

For the current known lower (resp. upper) bound of the variable x we write $\underline{x}$ (resp. $\overline{x}$). For convenience we define the *interval* I_i of variable x_i (with respect to a given constraint) as the contribution the variable makes to the "variability" of the range of the LHS of (1), i.e.

$$I_i = a_i(\overline{x_i} - \underline{x_i})$$

Following (loosely) [1], let

$$F = b - \sum_{i=1}^{n} a_i \underline{x_i} \tag{2}$$

Then the bounds consistency condition for the constraint for any x_j is

$$x_j \leq \frac{F}{a_j} + \underline{x_j} \tag{3}$$

Note that if $F < 0$ then the constraint is unsatisfiable (failure).

As noted in [1], (2) and (3) allow us to compute all the bounds imposed by the constraint in two passes over the constraint: one to compute F, and one to compute the (upper) bounds.

In order to maintain bounds consistency, this computation needs to be repeated whenever one or more lower bounds have changed. What we do in the following is to investigate cases where this computation step can be performed in sub-linear time.

3 Refinements

The refinements in this section are based on the observation that if

$$\overline{x_j} \leq \frac{F}{a_j} + \underline{x_j}$$

i.e.

$$I_j \leq F \tag{4}$$

then x_j is already bounds-consistent with respect to the constraint (c.f. (3)). In particular, if

$$F \geq \max_j I_j \tag{5}$$

then no bound updates will occur. Moreover, if it is not the case, then the variables affected are exactly those for which the condition (4) is violated.

There are several ways we might try to exploit this. One is to try to detect (cheaply) when no bound updates will occur. Another is to try to determine

(cheaply) which bounds need to be updated (which would yield the first case if there are none). Of course neither of these approaches need be implemented exactly: the right trade-off might be to use a (safe) approximation.

In order to achieve any significant benefit, any scheme ought to be able to achieve bounds consistency with respect to the constraint in sub-linear time in at least some cases; otherwise it will be at best a constant factor faster than basic two-pass propagation. In particular this means we cannot afford to recompute F every time we wish to propagate the constraint.

3.1 Maintaining F

F is actually quite easy to update incrementally: whenever the lower bound of a relevant variable (say x_j) is modified, simply adjust F accordingly:

$$F := F - a_j(new(\underline{x_j}) - old(\underline{x_j}))$$

Note that the cost of maintaining this incrementally is linear in the number of lower bound changes since the last time the constraint was propagated. It is possible that this could be more than the number of the variables in the constraint (making the cost super-linear). However, most solver implementations already incur a cost for notifying each relevant constraint on every bound update anyway (meaning that maintaining F incurs at most a constant factor penalty).

3.2 Perfect Propagation

Consider the set of variables x_j such that $F < I_j$. As noted earlier, these are exactly the variables which need to be updated to achieve bounds consistency. One way of quickly identifying these variables is to maintain a heap (priority queue) for every constraint, with one heap entry for each variable x_j, using the corresponding interval sizes I_j for ordering (largest I_j on top). This heap gives constant time access to the x_j with the largest I_j, and the variables needing a bound update are those at the top of the heap. Indeed, if there are p variables which need bound updates, these can be identified in $o(p)$ time.

The heap is an auxiliary data structure associated with the constraint, and maintained over the lifetime of the constraint, i.e. until the constraint is found to be entailed or disentailed. The heap can be set up in linear time (see [2]) during constraint set-up, but obviously needs to be maintained as the I_js shrink during the computation. Whenever this occurs (i.e. on every bound change), the variable's entry may need to be pushed down the heap, which is $O(\log n)$. These update operations need to be undone on backtracking, which can be done with the same (or perhaps better) complexity as the operations themselves.

Note that if the variables are real (non-integer) variables or the coefficients are unit, then the heap does not even need re-balancing after propagation since all the adjusted I_js are identical. If the variables are integer and have non-unit coefficients, then any bound update which involved rounding may result in a heap adjustment being necessary.

3.3 Reduced Frequency Propagation

The cost of maintaining the heap as discussed above may be prohibitively expensive. Fortunately, there are cheaper ways to exploit the condition (5). For example, we can just cache the value of $\max_j I_j$ from the last time the constraint was propagated and use this as an approximation of the actual current value of $\max_j I_j$. As long as F is no smaller than the cached value, no propagation is necessary. When it is smaller, then it may still be that no propagation is necessary, but we will not know without either recomputing $\max_j I_j$ or doing the propagation. This technique allows us to skip the propagation in some cases (regardless of whether this propagation was to be performed using the basic two-pass method or one of the other techniques described below).

We expect this to be of most benefit when the constraint is slack; that is, when F is significantly larger than $\max_j I_j$. In such situations, lower bounds (and thus F) may be updated many times before F becomes less than the cached value of $\max_j I_j$. Before that point, propagation is guaranteed not lead to any bound updates and can therefore be skipped safely.

3.4 Short-Circuit Propagation (I)

Another technique, which would be most effective when the variables are boolean and the coefficients vary, is to sort the constraint by decreasing I_i when it is first set up, recording this initial interval for each term. Then, when the constraint is being propagated, once a term is reached which has recorded interval no larger than F, there is no need to consider any remaining terms (because they are all guaranteed to satisfy (4)) and the propagation process may stop. This is particularly effective for booleans since a propagation pass fixes the values of the variables for some prefix of the terms in the constraints, and these terms do not need to be considered again: next time a propagation is required, it can continue from where it left off. Note that this means such a constraint can be propagated in $O(n)$ time amortised over a forward execution of the constraint — albeit after an initial setup time of $O(n \log n)$.

4 Entailment-based refinements

We now consider the issue of entailment. Let

$$E = \sum_{i=1}^{n} a_i \overline{x_i} - b$$

If we know E as well as F then there are further interesting things we can try. Observe that if $E \leq 0$ then the constraint is entailed and we need never consider it again (and need not maintain any information associated with it).

4.1 Basic Entailment Check

When using basic two-pass propagation, during the first pass to compute F, one can also compute E, thus enabling entailment to be detected.

4.2 Incremental Entailment Detection

When using one of the sub-linear propagation techniques from Section 3, we do not want to scan the whole constraint to check entailment as this would destroy the sub-linearity. To overcome this, we can maintain E in much the same way as we can F, by monitoring the relevant bounds and adjusting E accordingly when they change, keeping everything sub-linear.

4.3 Short-Circuit Propagation (II)

Next note that

$$E + F = \sum_i I_i \tag{6}$$

Now suppose that we have "propagated" some set of variables T (updating E appropriately), and that

$$\sum_{i \in T} I_i \geq E \tag{7}$$

Then we have that

$$\sum_{i \notin T} I_i \leq F$$

In particular,

$$I_i \leq F, \quad i \notin T$$

That is, we can skip propagating the rest of the constraint because none of the intervals are large enough to warrant adjusting.

Another way of looking at it is that setting all the variables in T to their lower bounds would result in the constraint being entailed, and so for the remaining variables all remaining elements of their domains are feasible (and hence cannot be pruned).

Note that most benefit can be derived from the condition (7) if we consider the variables with largest I_i first. As a heuristic we can sort the constraint by decreasing I_i when the constraint is set up (as in Section 3.4), and when propagating, consider them in this order. The rationale is that the smaller intervals at the end of the list cannot get any larger, so while the large ones at the front may get smaller, any large ones must still be towards the front of the list.

If one is also using reduced frequency propagation (Section 3.3) and thus caching $\max_j I_j$, then it may be useful to start the propagation with the corresponding variable x_j. In highly asymmetric constraints it may be that this interval alone is as large as E, meaning propagation can be stopped without looking at any other variables.

F	E	pre-sort I_js	I_js heap	Applicable techniques
no	no	no	no	Basic two-pass propagation Basic entailment checks
yes	no	no	no	Reduced freq. propagation Short-circuit entailment checks
yes	yes	no	no	Reduced freq. short-circuit (II) propagation Incremental entailment detection
yes	no	yes	no	Reduced freq. short-circuit (I) propagation Short-circuit entailment checks
yes	yes	yes	no	Reduced freq. short-circuit (I) & (II) propagation Incremental entailment detection
yes	yes	no	yes	Perfect propagation Incremental entailment detection

Table 1. Interesting combinations of techniques and their requirements

4.4 Short-Circuit Entailment

Note that we can also use (6) to derive a condition which allows us to short-circuit entailment checks (assuming we are not maintaining E incrementally). If we have processed a set of variables T (during a propagation pass or explicit entailment check) and we have

$$\sum_{i \in T} I_i > F$$

Then we have that

$$E > \sum_{i \notin T} I_i \geq 0$$

and hence the constraint cannot be entailed.

5 Discussion

Table 1 summarises the interesting combinations of the techniques presented in the earlier sections. We do not consider the other combinations interesting for the following reasons:

- All the propagation approaches discussed in this paper require F so that we know whether or not a bound we are looking at requires updating (and if so by how much). If we do not maintain it we must recompute it, which is $o(n)$; thus the basic two-pass method (also $o(n)$) is optimal if F is not maintained.
- If we are using a heap to manage the I_js, then since we need to check whether the heap needs adjusting on every bound update anyway, we might as well maintain E while we are at it: it is a constant extra cost per upper bound change, and it allows us to stop maintaining F and the heap once entailment occurs.

– Pre-sorting the constraint based on initial I_js not useful if we are going to maintain a heap.

5.1 Equations

Equations could of course be implemented using a pair of matched inequalities. However, it is obvious that this could be improved upon. To begin with, an equation is never entailed until all the variables have become ground. Thus there is no point checking for entailment (once everything is ground, there is no work to save).

Also, much computation can be shared between the two halves of the constraint. For instance, if a heap is being used to manage the I_js, then this heap is common. Also, one half's E is the other half's F, so that options such as short-circuit propagation (II) which depend on E can be employed without incurring any additional maintenance cost.

There is room for further exploitation here, but this is beyond the scope of this paper.

5.2 Heuristic Method Selection

It may be possible to select one of the above methods as being suitable for a particular constraint at the time the constraint is set up. For example, consider $\sum_{i=1}^{n} x_i \leq k$ where each x_i has domain $\{0,1\}$. If k is small then the constraint is tight, and it may not be worth checking entailment: $n-k$ upper bounds have to be reduced to 0 before entailment occurs, saving at most k checks of (5) on lower bound changes. On the other hand, if k is close to n then it probably is worth checking entailment: it introduces only $O(n-k)$ extra work but could save up to k checks of (5).

5.3 Specialisations

In a number of cases, further benefit could be obtained by specialising the above techniques. In particular, pseudo-boolean constraints seem good candidates for this, as do constraints with only unit coefficients.

Some such specialisations are obvious: for instance, if all coefficients are unit and all variables are boolean (meaning all intervals are of size 1) then there is no point having a heap, and the same effect can be achieved with a lower complexity data structure (a list). Another case, for booleans with arbitrary coefficients, relates to the method presented in Section 3.4: there is no need to record the initial interval for each variable, since this is the same as the variable's coefficient.

5.4 Floating point considerations

In principle, all of the above techniques (except where noted) work for both integer and real coefficients and variables. However, if floating point numbers

are being used to approximate reals, then floating point issues must be taken into account. Primarily, this involves making sure the results of floating point operations are rounded appropriately, so that (for example) F and E are always overestimated, so that (for example) bounds imposed are conservative and entailment is never detected when it shouldn't have been.

With the methods that update values incrementally, the errors in these values can increase as the computation progresses. To maintain accuracy, the values can be periodically re-computed from scratch.

6 Future Work

Future work includes examining the open questions discussed above. This will require further theoretical analysis of the alternative algorithms, as well as implementing some of the more interesting approaches. An evaluation based on these implementations should then provide insight into which techniques could be useful under what circumstances.

We would also like to do a comparison with other techniques for achieving bounds consistency for linear constraints, such as indexicals.

References

1. Warwick Harvey and Peter J. Stuckey. Constraint representation for propagation. In Michael Maher and Jean-Francois Puget, editors, *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming — CP'98*, LNCS 1520, pages 235–249. Springer, 1998.
2. Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, second edition, 1998.
3. J-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.
4. W. J. Older and A. Velino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 175–196. MIT Press, 1993.