Propagating equalities and disequalities

Neil Moore, supervised by Ian Gent and Ian Miguel

School of Computer Science, University of St Andrews, Scotland

1 Introduction

This paper describes an idea to improve propagation in a constraint solver. The most common frameworks for propagation are based on AC5 [5] where propagators can accept bound changed, assignment and value removal events. The idea is to extend this to allow constraints to generate and process equality and disequality events as well, e.g., $r \rightarrow x = y$ can generate an equality event whenever $r = true$, meaning that from now on in any solution $x = y$. Every constraint in the minion solver [3] can produce such events and benefit in some circumstances from receiving them.

In this paper I will give an example where the idea works successfully, describe the additional propagation that various propagators in minion can achieve, and describe implementation issues.

2 Example

Consider the following CSP: variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ each with domain $\{1, \ldots, n\}$ and constraints $x_1 = y_1$, $x_n \neq y_n$ and $\forall i$, $x_i = y_i \Leftrightarrow x_{i+1} = y_{i+1}$. Chris Jefferson has proved [6] that with any static variable ordering, it takes time exponential in n for backtracking search and propagation to prove that no solution exists. However by propagating equalities and disequalities the following will occur:

1. $x_1 = y_1$ will generate event $x_1 = y_1$
2. $x_1 = y_1 \Leftrightarrow x_2 = y_2$ will receive the event and produce $x_2 = y_2$
3. $\ldots$
4. $x_{n-1} = y_{n-1} \Leftrightarrow x_n = y_n$ will receive $x_{n-1} = y_{n-1}$ and produce $x_n = y_n$
5. constraint $x_n \neq y_n$ will recieve the event $x_n = y_n$ and thus fail

As I show in Section 4 this can be implemented in polynomial time and hence a significant speedup is achievable on this type of instance. I have implemented this idea and achieved exponential speedups as expected. I hope to be able to present practical results on this and other more realistic instances during my doctoral programme talk.

3 The technique

I propose to allow propagators to generate and receive events of the form $x = y$ and $x \neq y$ where x and y are variables. Propagators are not able to detect such events just by inspecting the solver state, because they may be true without being *entailed by the store*. For example, in a CSP containing $x = y$ with $\mathrm{dom}(x) = \mathrm{dom}(y) = \{1, 2, 3\}$ the propagator can remove no values. $x = y$ is true in any solution but not entailed by the constraint store. Conversely the AC5-style events can always be detected by propagators inspecting the domain state.

As a first example, I exhibit a rule for constraint *abs* (defined $x = |y|$):

Theorem 1. *In constraint $x = |y|$, if $x = y$ then we can infer that $y \geq 0$.*

Proof. $y = x = |y| \implies y = |y| \implies y \geq 0$.

The standard propagator for $x = |y|$ will prune so that $x \geq 0$ anyway, and this fact combined with the event $x = y$ itself subsumes $y \geq 0$. However I will shortly show that the contrapositive of the rule is useful. As it happens, the converse of the rule is also true:

Theorem 2. *In constraint $x = |y|$, if $x \neq y$ then we can infer that $y < 0$.*

Proof. $|y| = x \neq y \implies |y| \neq y \implies y < 0$.

These theorems show that *abs* can benefit from receiving (dis-)equality events, allowing it to prune all negative (positive) values from the domain of y. Can it also produce such events? I now exhibit a simple meta-theorem to show that any constraint that can use (dis-)equalities can also produce them for other constraints to use:

Theorem 3. *In constraint C, if event* allows us to infer *condition*, then $\neg$*condition* allows us to infer $\neg$*event*.

Proof. Omitted.

For example, a rule to *produce* events from Theorem 1 can be obtained:

Corollary 1. *In constraint $x = |y|$, if $y < 0$ then we can infer that $x \neq y$.*

Proof. Immediate from Theorems 1 and 3.

In outline, in my new propagation framework a constraint may choose to receive (dis-)equality events in addition to the normal set of pruning, assignment and bound events. It can also notify other constraints that it has inferred a (dis-)equality by generating an event. The solver will

- add the event to the store; other constraints can check for (dis-)equalities (analogous to variable checks like inDomain, getMin, etc.)
- pass on (dis-)equalities to constraints that have registered in interest (analogous to static and watched triggers on value removals); and
- propagate the (dis-)equality constraint itself (analogously to how producing the AC5 event $x \not\leftarrow 1$ removes 1 from x's domain).

I have analysed the whole set of constraints in minion, and found that every single one can benefit from (dis-)equality propagation. These have not been chosen specially. A selection are tabulated in Table 1 without proof. For example, Theorems 1 and 2 are reproduced as the first and second lines of the table. I am unable to give the complementary rules by Theorem 3 for space reasons. I attempt to only give rules that are not currently captured by the minion propagation algorithms.

Constraint	Definition	Condition	Event	Notes
abs	$x = \|y\|$	$x = y$	$y \geq 0$	See Theorem 1. Subsumed by other propagation.
abs	$x = \|y\|$	$x \neq y$	$y < 0$	See Theorem 2
alldifferent		none	$v_i \neq v_j$	Generate all disequalities.
difference	$z = \|x - y\|$	$z \neq 0$	$x \neq y$	
difference	$z = \|x - y\|$	$z = 0$	$x = y$	
diseq	$x \neq y$	none	$x \neq y$	
div	$z = \lfloor x/y \rfloor$	$z \neq 1$	$x \neq y$	
element	$vec[i] = e$	$i = idx$	$vec[idx] = e$	
eq	$x = y$	none	$x = y$	
gcc	Too complex to describe here, see Section 3.1.			
ineq	$x \leq y + c$	$x = y$	$c \geq 0$	
ineq	$x \leq y + c$	$x = c$	$y \geq 0$	
lexleq	Too complex to describe here, see Section 3.2.			
lexless	Similar to lexleq.			
max	$x = \max(y, z)$	$y = z$	$x = y$	Also $x = z$
max	$x = \max(y, z)$	$x \neq y$	$x = z$	
max	$x = \max(y, z)$	$x \neq z$	$x = y$	
modulo	d/e=? rem r	$d = e$	$r = 0$	
modulo	"	$e = r$	fail	Remainder must be in the range $0 \ldots e - 1$
modulo	"	$d = r$	fail	Remainder must be less than dividend
pow	$x^y = z$	$x \neq z$	$y \neq 1$	
pow	$x^y = z$	$x = z$	$y = 0 \vee y = 1$	
product	$xy = z$	$x = z$	$y = 1$	Also $y = z$
product	$xy = z$	$x \neq z$	$y \neq 1$	Also $y = z$
table	See Section 3.3			
OMITTED	and, or, reify, reifyimply metaconstraints; min; watchvecneq; minuseq; occurrence; sumleq; sumgeq; weightedsum			

Table 1. Propagation rules

3.1 Global cardinality constraint

The global cardinality constraint (GCC) [8] given vector of variables $v_1, \ldots, v_m$, count variables $c_1, \ldots, c_n$ and constants $val_1, \ldots, val_n$ ensures that there are c_i occurrences of val_i in $v_1, \ldots, v_m$.

Without being specific about the propagation algorithm, additional propagation could be achieved in the following case, for example:

1. Say that $dom(v_1)$, $dom(v_2)$ and $dom(v_3)$ were all $1, 2, 3$, and 1 doesn't appear in any other domains.
2. Furthermore, 1 has to be repeated twice in $v_1, \ldots, v_m$.
3. It is easy to tell that either v_1 or v_2 has to be 1, by the pigeonhole principle.
4. If we now know that $v_1 = v_2$ then we can tell straight away that $v_1 = v_2 = 1$ and $v_3 \neq 1$.

Dis-equalities can also be exploited:

1. Say that $dom(v_1)$ and $dom(v_2)$ are both $1, 2$, and 1 doesn't appear elsewhere.
2. Furthermore 1 can be repeated either once or twice.
3. If we now know that $v_1 \neq v_2$ then straight away we know that 1 can't be repeated twice.

It is comparatively simple to incorporate disequality information into the gcc algorithm, by using an alternate network flow design instead of the standard one described by Régin [8]. I haven't yet found an efficient design that incorporates equality information. I also haven't tried to make gcc produce (dis-)equalities although it must be possible.

3.2 Lexicographical ordering constraint

The lexicographical ordering constraint ensures that $x_1 \ldots x_r \leq_{lex} y_1 \ldots y_r$, e.g., $100 \leq_{lex} 101$ but $101 \not\leq_{lex} 100$. The standard literature algorithm obtaining GAC is given in [1]. It works by maintaining two indices α and β. α is the index such that all more significant indices are assigned to the same value. β is the most significant index such that the tail of the vectors from that position must violate the constraint. For example, the following table gives the current domains of the variables in the lexleq constraint.

i	0	1	2	3	4
$dom(x_i)$	2	$1,3,4$	$2,3,4$	1	$3,4,5$
$dom(y_i)$	2	1	$1,2,3$	0	$0,1,2$

In this case $\alpha = 1$ because x_0 and y_0 are assigned the same; $\beta = 3$ because any assignment now has $x_3 > y_3$ and $x_4 > y_4$. The propagator maintains these values α and β as domains are narrowed. If $\alpha > \beta$ the constraint fails. If $\alpha + 1 = \beta$ the propagator enforces $x_\alpha < y_\alpha$. If $\alpha + 1 < \beta$ then the propagator enforces $x_\alpha \leq y_\alpha$.

It is quite easy to see how to incorporate (dis-)equality information into the algorithm: If we find out that $x_\alpha = y_\alpha$ then α can be incremented. Once $\alpha + 1 = \beta$ produce the event $x_\alpha \neq y_\alpha$.

3.3 Table constraint

Roughly speaking, the table constraint propagator in minion works by finding a *valid* supporting tuple for each variable/value pair. To take account of (dis-)equality information the criteria for "validity" is changed. Before, it was that each variable/value pair in the tuple is in its respective domain. After, tuples must also conforms to any and all known (dis-)equalities. For example, tuple $(x = 1, y = 1, z = 2)$ is disallowed if we know $x \neq y$ and/or if we know $y = z$.

Conversely, if all tuples whose components are in their respective domains are such that a particular pair of components are always equal or unequal then the corresponding event can be generated.

4 Implementation

(Dis-)equalities can be handled in a similar way to other propagation events like $v \leftarrow 1$ or $v \not\leftarrow 3$. Note that there are a quadratic number of different (dis-)equality events possible, the same as (dis-)assignment and bound changed events.

Constraints should have the following facilities available to them:

- Set up static triggers on chosen (dis-)equality events (if the (dis-)equality is already true, it should trigger immediately after being set up).
- Generate (dis-)equality events for chosen pairs of variables.
- Check if an equality is true, false or unknown.

(Dis-)equalities should be removed after backtrack. If $x = y$ and $x \neq y$ are both generated the solver should fail and backtrack. If $x = y$ and $y = z$ are generated, the solver should ensure $x = z$ is generated.

(Dis-)equalities need only be generated explicitly by propagators and when they are not obvious from the variable domains. For example, even if $x = 1$ and $y = 1$ event $x = y$ may not result. This property is not essential, and it would be a good idea to try generating all events. The advantages are as follows:

- Events that are obvious from the variable domains should be picked up in the normal course of constraint propagation. It is better for (dis-)equality propagation to be orthogonal to normal propagation so it may be easily switched off.
- Avoids introducing vast numbers of events for problems with small domains where equalities are likely, for example in a boolean problem. Also, no matter the domains, as more variables are assigned more spurious equalities and disequalities are produced.

The disadvantage is that this will lead to code duplication: a propagator might like to use disequality as a trigger, so that it will only propagate when it receives such an event. However using this framework it cannot assume that it will get the (dis-)equality trigger and hence it must place other triggers.

It would be sensible at some point to evaluate whether generating all possible (dis-)equalities is advantageous.

Implied events If events $x = y$ and $y = z$ are both produced, then $x = z$ is implied. The solver should ensure that these implied events are created, because the variable y may not be known to a propagators with a trigger on $x = z$. Implied disequality events also exist. Standard algorithms using the union-find data structure can be exploited to solve these problems, see [7].

When to enable Potentially, enabling (dis-)equality propagation could be damaging to propagation speed, if the dynamic characteristics of the problem are not suitable. By using a similar technique to [9] the solver can detect cases when (dis-)equality propagation will be unsuccessful. Details are omitted due to space considerations.

5 Previous work

This idea is so simple that it is inevitable that similar work will exist. The aim of the work is to be simple and effective; hence I do not seek to compete with more general ideas. Furthermore, I have tailored the idea to work in a *propagation solver*, so it is not coincident with similar ideas in other types of automated search and reasoning. I would be interested to hear of any other related work.

Hägglund [4] has worked on allowing arbitrary constraints to be put in the store. This clearly generalises (dis-)equality propagation. My work is on a special case of this idea chosen to be efficient and more common in practice. Constraint handling rule (CHR) solvers [2] can easily incorporate (dis-)equalities into their rules. However CHR rules are not suitable for replacing constraint propagators for reasons of efficiency. Some satisfiability modulo theories (SMT) solvers use the theory of *equalities with uninterpreted functions* [7]. Although (dis-)equality propagation applies to this theory and is exploited, the theory is limited in its expressivity compared to what's available in a CSP solver. Some CSP solvers are able to unify variables, meaning that once they are detected to be equal they become the same variable; Eclipse is an example of such a solver. This allows equality propagation to be implemented, but does not include disequalities. Theorem 3 shows that both are necessary to achieve full propagation.

References

1. Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Propagation algorithms for lexicographic ordering constraints. *Artif. Intell.*, 170(10):803–834, 2006.
2. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
3. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.
4. Bjorn Hagglund. A framework for designing constraint stores. Master's thesis, Linkoping University, 2007.
5. Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.
6. Chris Jefferson, June 2009. Personal correspondence.
7. R. Nieuwenhuis and A. Oliveras. Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In G. Sutcliffe and A. Voronkov, editors, *12h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.
8. Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI/IAAI, Vol. 1*, pages 209–215, 1996.
9. Christian Schulte and Peter J. Stuckey. Dynamic analysis of bounds versus domain propagation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Twenty Fourth International Conference on Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 332–346, Udine, Italy, December 2008. Springer-Verlag.