Implementing Logical Connectives in Constraint Programming

Christopher Jefferson†, Neil Moore†, Peter Nightingale†, Karen E. Petrie*

* School of Computing, University of Dundee, UK
email: kpetrie@computing.dundee.ac.uk
† School of Computer Science, University of St Andrews, UK,
email: caj@cs.st-andrews.ac.uk, ncam@cs.st-andrews.ac.uk, pn@cs.st-andrews.ac.uk

Abstract

Combining constraints using logical connectives such as disjunction is ubiquitous in constraint programming, because it adds considerable expressive power to a constraint language. We explore the solver architecture needed to propagate such combinations of constraints efficiently. In particular we describe two new features (named *satisfying sets* and *constraint trees*) of the Minion solver [1]. We also make use of *watched literals* [2], and with these three complementary features we are able to make considerable efficiency gains.

A key reason for the success of Boolean Satisfiability (SAT) solvers is their ability to propagate OR constraints efficiently, making use of watched literals. We successfully generalise this approach to an OR of an arbitrary set of constraints, maintaining the crucial property that at most two constraints are active at any time, and no computation at all is done on the others. An AND algorithm is also given, and may be embedded within the OR. Using this approach, we demonstrate speedups of over 10,000 times in some cases, compared to traditional constraint programming approaches. We also prove that the OR algorithm enforces generalized arc consistency (GAC) when all its child constraints have a GAC propagator, and no variables are shared between children. By extending the OR propagator, we present a propagator for ATLEASTK, which expresses that at least k of its child constraints are satisfied in any solution.

Some logical expressions (e.g. exclusive-or) cannot be compactly expressed using AND, OR and ATLEASTK. Therefore we investigate *reification* of constraints. We present a fast generic algorithm for reification using satisfying sets and watched literals. In contrast to AND, OR and ATLEASTK, reification allows for any logical expression. We compare algorithms which use satisfying sets and watched literals with alternatives using static triggers, and again we demonstrate huge speedups.

1 Introduction

Problems often consist of choices. Making an optimal choice which is compatible with all other choices made is difficult. *Constraint programming* (CP) is a branch of Artificial Intelligence, where computers help users to make these choices. Constraint

programming is a multidisciplinary technology combining computer science, operations research and mathematics. Constraints arise in design & configuration, planning & scheduling, diagnosis & testing, and in many other contexts. This means that constraints are a powerful and natural means of knowledge representation and inference in many areas of industry and academia.

A *constraint satisfaction problem* (CSP [3]) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. For example, the problem might be to fit components (values) to circuit boards (decision variables), subject to the constraint that no two components can be overlapping. An assignment maps a variable to a value from its domain. Each constraint specifies allowed combinations of assignments of values to a subset of the variables. A *solution* to a CSP is an assignment to all the variables which satisfies all the constraints. Solutions are found for CSPs through backtrack search with an inference step at each node [3].

Modelling is the process of representing a problem as a CSP. To allow natural modelling of some problems, the logical connectives of AND and OR are required between constraints. For example, in our circuit board example you may have either component 1 OR (component 2 AND component 3) are against one edge. It is also sometimes useful to be able to apply NOT to a constraint, this is often done in CSP by means of *reification*. The reification of a constraint C produces another constraint C_r, such that C_r has an extra Boolean variable r in its scope, and (in any solution) r is set to true iff the original constraint C is satisfied. In this paper we discuss the understudied area of how to efficiently implement these logical connectives across constraints, which are the fundamental building blocks of CSP models [4] (chapter 11).

During the search for a solution of a CSP, constraint *propagation* algorithms are used. These propagators make inferences, recorded as domain reductions, based on the domains of the variables constrained. If at any point these inferences result in any variable having an empty domain then search backtracks and a new branch is considered. Propagators and generalized arc consistency (GAC) are important concepts in this paper. When considering a single constraint C, GAC is the strongest possible consistency that a propagation algorithm can enforce. Enforcing GAC removes all domain values which are not compatible with any solution of C. Bessiere defines GAC and discusses the complexity of enforcing it [4] (chapter 3).

In this paper we consider propagating logical combinations of constraints. For example, for constraints C_1, C_2, C_3, C_4 we may wish to post the following expression and propagate it efficiently.

$$(C_1 \wedge C_2) \Rightarrow (C_3 \vee C_4)$$

It is desirable to make use of existing propagators for C_1, C_2, C_3 and C_4 since these may be highly efficient specialised propagators.

1.1 A Traditional Approach

A traditional approach (probably the most common) is to individually create *reified* propagators for the four constraints. These introduce an additional Boolean variable representing the truth of the constraint (e.g. the reified form of C_1 is the constraint

$r_1 \Leftrightarrow C_1$, so in any solution r_1 is TRUE iff C_1 is satisfied). The logical expression is posted on the additional Boolean variables. The example above translates into the following collection of constraints. (In some solvers it would be necessary to further decompose $(r_1 \wedge r_2) \Rightarrow (r_3 \vee r_4)$.)

$$r_1 \Leftrightarrow C_1, \quad r_2 \Leftrightarrow C_2, \quad r_3 \Leftrightarrow C_3, \quad r_4 \Leftrightarrow C_4, \quad (r_1 \wedge r_2) \Rightarrow (r_3 \vee r_4)$$

This scheme has two major disadvantages. First, it can be very inefficient because every reified constraint is propagated all the time. For example consider an OR of a set of n constraints. As we will demonstrate in Section 4, at most two constraints need to be actively checked at any time. However, a reification approach will propagate all n reified constraints at all times. Second, developing reified propagators individually for each constraint is a major effort. We address both issues in this paper.

1.2 Two Vital Features of a Solver for a New Approach

The key finding of this work is that two vital features of the solver must be combined to achieve efficient propagation of logical connectives. If either feature is not available, then the other is of limited benefit. The two features are *constraint trees*, which allow a *parent constraint* to control the propagation of its children, and *movable triggers* which allow a constraint to change the events it is interested in during search.

Consider an OR of n constraints over disjoint scopes. We will show that at most two of the constraints need to be considered at any time, because if two of the constraints are satisfiable then no propagation can occur. Once two satisfiable constraints have been identified, all other constraints are presently irrelevant and no computation time should be wasted on them. This is essential to efficiency when n is large.

Constraint trees allow us to stop checking irrelevant constraints. However, this is not enough to achieve zero cost for irrelevant constraints: there is a cost to generate trigger events for the constraints. It is necessary to remove triggers not currently of interest, hence movable triggers are also required.

The following table summarises the costs caused by irrelevant constraints.

	Static Triggers	Movable Triggers
Reification	All reified constraints propagated at all times	All reified constraints propagated at all times
Constraint Trees	Trigger events received for all constraints at all times	Irrelevant constraints cause *no cost*

1.3 Overview

First we give a detailed motivating example in Section 1.4. Following this we explore previous work in the area in Section 2.

There are a number of solver architecture decisions which impinge on propagating logical combinations of constraints. In Section 3 we describe three architecture features which are key to the new algorithms presented in this paper. Two of the features (*satisfying sets*, Section 3.3 and *constraint trees*, Section 3.2) are novel to the best of our

knowledge. Watched literals [2] are also described in Section 3.1 to aid understanding of the rest of the paper.

In Section 4, we present propagators for the operators OR, AND and ATLEASTK (which ensures that at least k of a set of constraints are satisfied in any solution) over sets of constraints. Via the constraint trees framework, each OR, AND and ATLEASTK is itself a constraint, and may be a child of some other constraint. Therefore OR, AND and ATLEASTK may be nested to any depth, and also may be reified using the algorithms given in Section 5. The propagator for OR is inspired by unit propagation (using watched literals) in SAT [5], and maintains the crucial property that only two child constraints are checked (or one propagated) at any time — no computation at all is done on the others. The algorithm is named Watched OR. The Watched ATLEASTK propagator is presented for ATLEASTK, with similar properties to Watched OR. A simple propagator for AND is also presented. Section 4 also contains experiments on the efficiency of Watched OR, AND and ATLEASTK, which demonstrate huge speedups in some cases.

In Section 5 we consider reification. Some logical expressions (e.g. exclusive-or) cannot be compactly expressed using only AND, OR and ATLEASTK, so a more general approach is needed. Therefore we investigate the use of satisfying sets, watched literals and constraint trees for reification of constraints. To avoid implementing reified propagators for individual constraints, we developed four generic algorithms which can be used with any constraint C, provided that there is a propagator for $\neg C$ available. We compare algorithms which use satisfying sets and watched literals with alternatives using static triggers, and again we demonstrate huge speedups in some cases.

Finally, the paper is concluded in Section 6.

1.4 Motivating Example

The following example was proposed by a user of the Minion solver [1]. It is a very simple CSP, but it shows the need to combine constraints logically. Two models are given using existing constraints from the Minion solver, but in both cases there are significant disadvantages. This motivates a new approach, the Watched OR algorithm presented in Section 4.

We have two arrays of variables, X and Y of equal length n, which are constrained to be different (i.e. $\exists i.\ X[i] \neq Y[i]$).

The first model introduces auxiliary variables (an array of Booleans, B of length n). It makes use of two constraints: $r \Leftrightarrow x_1 \neq x_2$ and $\Sigma(B) \geq 1$, both of which are available in Minion.

$$\forall i.\ X[i] \neq Y[i] \ \Leftrightarrow B[i], \quad \Sigma(B) \geq 1$$

Propagation (using the standard Minion propagators) of this model is equivalent to expressing 'vector not-equal' as a single constraint, and enforcing GAC on that constraint. However, our experiments will show that this method performs poorly (Section 4.6.1).

A second model uses the **element** constraint. The **element** constraint takes an array of variables X and individual variables y and z. In any solution, z takes the value at position y in array X (i.e. $X[y] = z$). An efficient GAC propagator for the **element** constraint was devised by Gent et al. [2], using watched literals for propagation.

Watched literals are explained in more detail in Section 3.1. The second CSP model is as follows.

$$X[u] = v, \quad Y[u] = w, \quad v \neq w$$

This model runs faster then the previous model in terms of nodes per second. However, it does not obtain the equivalent of GAC propagation, so the search space explored can be far larger then that of the previous model. This is explained in more detail in Section 4.5.

It would be natural to express the existential as an OR, and if this could be propagated directly (achieving GAC) it would avoid the overhead of additional variables and the disadvantages of the **element** model.

$$(X[1] \neq Y[1]) \vee (X[2] \neq Y[2]) \vee \ldots \vee (X[n] \neq Y[n])$$

By exploiting satisfying sets, constraint trees and watched literals, the Watched OR algorithm (presented in Section 4) is able to enforce GAC on the entire OR (because no variables are shared between the $\neq$ constraints). Watched OR is shown experimentally to dominate both the other models presented above, in Section 4.6.1.

2 Background

2.1 Preliminaries

A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is defined as a set of n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, a set of domains $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ where $|D_i| < \infty$ is the finite set of all potential values of x_i, and a conjunction $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints.

Within CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, a constraint $C_k \in \mathcal{C}$ consists of a sequence of $r > 0$ variables $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ with respective domains $\mathcal{D}_k = \langle D_{k_1}, \ldots, D_{k_r} \rangle$ s.t. $\mathcal{X}_k$ is a subsequence[1] of $\mathcal{X}$, $\mathcal{D}_k$ is a subsequence of $\mathcal{D}$, and each variable x_{k_i} and domain D_{k_i} matches a variable x_j and domain D_j in $\mathcal{P}$. C_k has an associated set $C_k^S \subseteq D_{k_1} \times \ldots \times D_{k_r}$ of tuples which specify allowed combinations of values for the variables in $\mathcal{X}_k$.

A constraint is *satisfied* under a complete assignment to the variables iff the values of $\mathcal{X}_k$ in sequence form a tuple in C_k^S. The reified form ($r_k \Leftrightarrow C_k$) of a constraint C_k is satisfied iff $r_k \mapsto 1$ and C_k is satisfied, or $r_k \mapsto 0$ and C_k is not satisfied. The reifyimplied form ($r_k \Rightarrow C_k$) of C_k is satisfied iff $r_k \mapsto 0$, or $r_k \mapsto 1$ and C_k is satisfied.

The AND of a set of constraints is satisfied iff all constraints in the set are satisfied. The OR of a set of constraints is satisfied iff at least one of the constraints in the set is satisfied. The ATLEASTK of a set of constraints (with parameter k) is satisfied iff at least k of the constraints in the set are satisfied.

A *sub-domain* for a variable x is a subset of its domain. A list of sub-domains $\langle D_1, \ldots, D_k \rangle$ *allows* the list of assignments $D_1 \times \ldots \times D_k$. A *propagator* for a constraint C is a function which takes a sub-domain of each variable in $\mathcal{X}_C$ and returns a new list of sub-domains, which do not allow any extra assignments, and do not remove

[1] We use subsequence in the sense that $\langle 1, 3 \rangle$ is a subsequence of $\langle 1, 2, 3, 4 \rangle$.

any assignments which satisfy C. A propagator is *GAC* if it removes every domain value possible without removing an assignment which satisfies C. A complete discussion of propagators can be found in [4] (chapter 3).

One issue which is often ignored when discussing propagation algorithms is repeated variables. Usually propagation algorithms which achieve GAC will not achieve GAC when variables are repeated. For example many constraint solvers have a GAC propagator for $x \neq y$, but will not fail instantly when given the unsatisfiable constraint $x \neq x$. All the proofs in this paper assume that there are no repeated variables, which includes no sharing between child constraints. As with all propagators, the algorithms presented in this paper still provide correct propagators if variables are repeated, but in general will not achieve GAC.

2.2 Related Work on OR

Many authors have considered *constructive disjunction* for propagating OR. For example, Müller and Würtz [6, 7] present a constructive disjunction algorithm implemented in Oz. Assuming that all child constraints have GAC propagators, constructive disjunction is able to enforce GAC over the OR, regardless of whether child constraints share variables. However, this is achieved by making a copy of the variable domains for each child constraint and propagating each child independently. A value which is pruned by every child constraint (i.e. pruned in each copy of the domains) is then pruned globally by the OR. It is not clear that this algorithm can be implemented efficiently. Lagerkvist and Schulte [8] observed a performance penalty of over 45% when executing propagators on copies of the domains, and mirroring the result back to the primary domains.

Constructive disjunction may be valuable for problems where strong propagation of OR is required. However, in this paper we consider more lightweight methods that do not require duplication of variable domains. Therefore we consider constructive disjunction to be outside the scope of this paper.

Bacchus and Walsh [9] give some theoretical results about logical combinations of constraints, including AND, OR and negation. Concerning OR, the paper only states that the set of inconsistent values of the OR is the intersection of the inconsistent values of each child constraint. This would perform the same domain reductions as constructive disjunction. The authors give a basic algorithm but do not consider incremental propagation (which is vital for efficiency). Adapting this algorithm for incrementality would require tracking the state of variable domains independently for each child — essentially duplicating the variable domains. This would be equivalent to the algorithm of Müller and Würtz [6, 7].

Lhomme [10,11] presents an alternative to constructive disjunction which performs the same domain reductions. Lhomme's algorithm is claimed to be more efficient than constructive disjunction. It is based on finding satisfying assignments (represented as tuples of values) for the constraints. Each relevant variable-value pair is *supported* by a satisfying tuple for one of the constraints in the disjunction, or it is pruned.

While Lhomme's algorithm may be faster than constructive disjunction, it maintains a large set of supporting tuples (one for each variable-value pair where the variable is shared between two child constraints). Our proposed algorithm maintains only

two partial tuples (and enforces a weaker consistency), therefore it is much more lightweight.

2.3 Related Work on Reification

We focus on generic approaches to reification that can be applied to any constraint that has the appropriate algorithms defined for it. For example, we prove that a generic reification algorithm that enforces GAC efficiently requires GAC propagators for both the constraint and its negation.

Indexicals (proposed by Van Hentenryck et al. [12]) allow simple propagators to be specified in a high-level language. They can be extended slightly to allow reification [4] (Section 14.2.6). However, it is not possible to express global propagators such as AllDifferent [13] in the indexicals language.

Propia [14] allows constraints to be expressed as Prolog predicates. The predicate specifies the constraint semantically as opposed to giving a propagator for the constraint. To implement reification, a predicate would be required for both the constraint and its negation. Similarly to indexicals, it is not possible to specify sophisticated propagators in propia, therefore it does not offer an efficient generic solution.

Schulte proposed a generic reification algorithm [15] based on the concept of *computation spaces*. A computation space is an isolated environment which allows a propagator to be executed without affecting the primary variables. The space includes duplicate variables. For $r_i \Leftrightarrow C_i$, C_i is posted in the space, and propagated. If it fails, then $r_i \neq 1$ (i.e. 1 is pruned from r_i). If it is entailed (i.e. equivalent to the constraint TRUE), then $r_i \neq 0$. If $r_i = 1$ then the effects of propagating C_i are copied to the primary variables. In the case where $r_i = 0$, there is no propagation of $\neg C_i$, and the algorithm does nothing until C_i is entailed.

The approach later proposed by Lagerkvist and Schulte [8] is virtually the same algorithm implemented with propagator groups. The only apparent difference is that value removals in the primary variables are copied to the duplicate variables, enabling incremental propagation of C_i.

Both these approaches have the disadvantage that they duplicate variables. Lagerkvist and Schulte compared a hand-implemented reified constraint to the generic algorithm. The generic algorithm was substantially slower, with the solver taking between 29% and 106% extra time [8].

The commercial product ILOG Solver implements reification, but we found no literature describing the algorithm.

In Section 5 we propose new reification algorithms which avoid the overhead of duplicating variables, while also being able to encapsulate any propagator, unlike indexicals or propia.

3 Solver Architecture

In order to implement logical connectives efficiently, we made a number of solver architecture decisions which are described in this section.

3.1 Watched Literals

One important part of how propagators are implemented is how they are called. Almost all solvers allow constraints to attach *triggers* to variables, which denote that this constraint should be informed when a variable domain is changed. When these triggers are activated they are placed on a queue. The solver then moves through this queue, calling each constraint in turn.

There are many modifications and extensions to this basic principle. Rather then a constraint being informed whenever a variable domain is changed for example, it could instead only be informed if a particular domain value is removed. Here we are concerned only about the triggers themselves, rather then what happens once they are triggered. In Minion there are three classes of triggers, outlined below and discussed in depth in [2].

Static: These triggers are placed on variables at the beginning of search. They can never be moved or removed.

Backtracking: These triggers can be placed, moved and removed during search. When search backtracks, they are restored to their previous location.

Watched: These triggers can be placed, moved and removed during search. When search backtracks, they are **not** restored to their previous place.

The class of triggers considered in this paper are *watched* triggers. Using these triggers can produce great improvements in the performance of the solver, as there is no need to specially handle them when search backtracks. However, the fact that they can be moved and do not revert to their original position when search backtracks introduce several complications to the implementation of algorithms which use them. Example 1 demonstrates how watched literals are traditionally used to implement SAT.

Example 1. *This Example illustrates a search tree where watched literals are used to implement propagation for the SAT constraint $A \vee B \vee C \vee D$. This algorithm is based around the principle that as long as two clauses could be assigned true, no propagation can occur. Once only one clause is satisfiable, it must be true. Therefore at all points the algorithm watches two clauses, and when only one still holds it is assigned true.*

Domains				*Watch*		*Description*
A	*B*	*C*	*D*	*1*	*2*	
{0,1}	{0,1}	{0,1}	{0,1}	A	B	*Algorithm Setup*
{0,1}	{0,1}	{**0**}	{0,1}	A	B	*No effect*
{**0**}	{0,1}	{0}	{0,1}	C	B	*B triggered - Watch Moved to D*
{0}	{**1**}	{0}	{0,1}	D	B	*B triggered - Watch Left at B*
Search Backtracks to start for unrelated reason						
{0,1}	{0,1}	{0,1}	{0,1}	D	B	*No effect - triggers left on D and B*
{**0**}	{0,1}	{**0**}	{0,1}	D	B	*No effect*
{0}	{**0**}	{0}	{1}	D	B	*B triggered - D assigned.*
Search Backtracks for unrelated reason						
{0}	{0,1}	{0,1}	{0}	D	B	*Triggers on B and D become valid again.*

There are a number of important points to notice about Example 1. When the algorithm says that there is "no effect", there really is no effect at all. In particular, given a SAT constraint where the watches are placed on variables which are never propagated during search, the constraint is never considered during search, although obviously it does take up a small amount of memory. Also, when search backtracks the watches remain in situ. However, it is easy to see from this example that these new values provide a valid support.

3.2 Constraint Trees

All the following algorithms use the concept of *parent* and *child* constraints, where the parent constraint controls when a child is propagated, and crucially must be able to tell when child constraints are *disentailed* (i.e. when there is no assignment that satisfies the constraint given the current domains). This concept is very flexible, and we use it for disjunction of a set of constraints, at-least-n of a set of constraints, and reification.

Any constraint which has the appropriate procedures may function as a child constraint, hence it is possible to build a tree of constraints. This is useful to nest operators, constructing for example a disjunction of conjunctions of constraints. All the parent constraints we describe in this paper can also function as a child of another constraint.

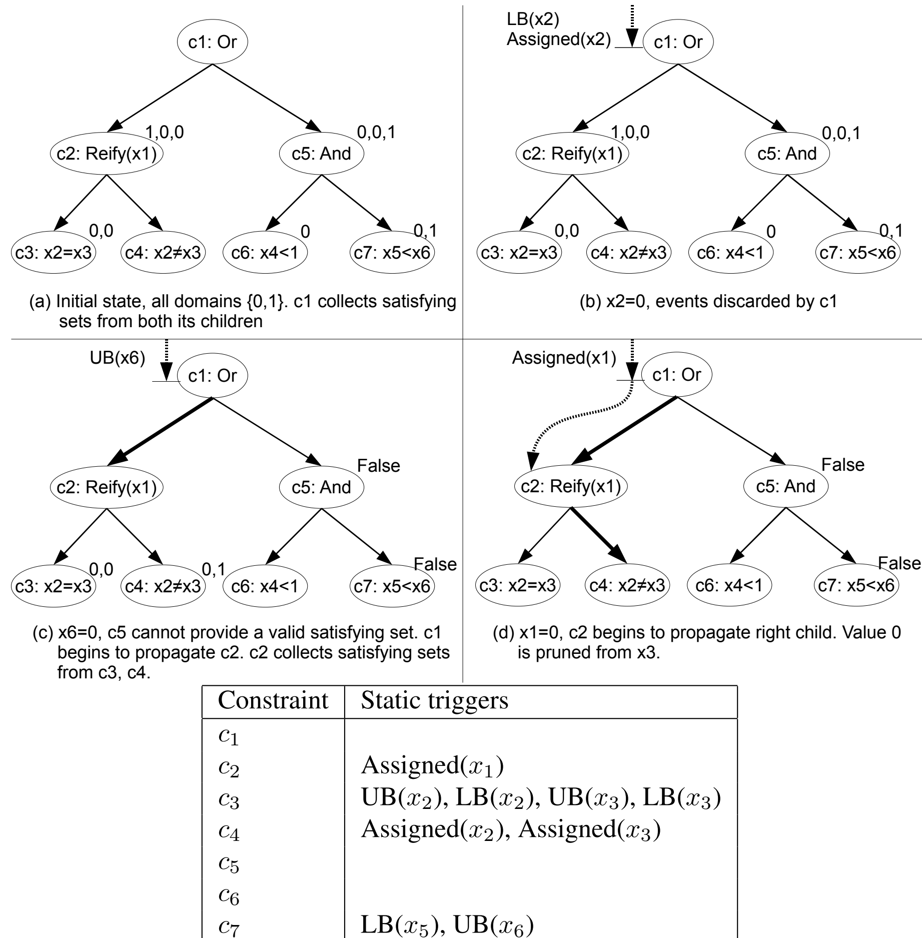
Static triggers are handled as follows in a tree of constraints. At setup time, all constraints in the tree place the static triggers that they need. During search, all trigger events are passed to the topmost constraint. Each parent constraint passes the appropriate trigger events through to the children which are currently propagating, and discards others. An example of this is shown in Figure 1. Three assignments occur in sequence ($x_2 = 0$, $x_6 = 0$ and $x_1 = 0$) and the corresponding events are passed to c_1 by the solver core. In Fig 1(b), c_1 is propagating neither of its children so it discards the two events. In (c), c_1 is propagating its left child, but the trigger event belongs to the right child so it is discarded. In (d), c_1 passes the trigger event on to c_2 because c_2 is currently propagating.

Movable triggers (i.e. backtracking or watched triggers) are somewhat more complicated, but they allow triggers for non-propagating children to be removed, reducing the number of unnecessary trigger events. Operations on movable triggers are described in detail with the algorithms in sections 4 and 5.

For both classes of trigger, the trigger events are passed in at the top of the tree, and filter down. This inevitably adds some overhead to propagating the constraints at the leaves of the tree. However we expect constraint trees to be shallow, limiting the overhead.

3.3 Satisfying Sets

In many of the algorithms in this paper, we will frequently want a fast method of checking if a constraint is satisfiable. One way of doing this is to execute its propagator and check to see if removes all the values from the domain of any variable. This is difficult to do with incremental propagators, as we must either ignore the incrementality, or keep a list of all the propagation the propagator would have made, for later nodes. In this section, we introduce satisfying sets, a simple and efficient framework which



(a) Initial state, all domains {0,1}. c1 collects satisfying sets from both its children

(b) x2=0, events discarded by c1

(c) x6=0, c5 cannot provide a valid satisfying set. c1 begins to propagate c2. c2 collects satisfying sets from c3, c4.

(d) x1=0, c2 begins to propagate right child. Value 0 is pruned from x3.

Constraint	Static triggers
c_1	
c_2	Assigned(x_1)
c_3	UB(x_2), LB(x_2), UB(x_3), LB(x_3)
c_4	Assigned(x_2), Assigned(x_3)
c_5	
c_6	
c_7	LB(x_5), UB(x_6)

Figure 1: Example of static trigger events and satisfying sets in a constraint tree representing $(x_1 \leftrightarrow x_2 = x_3) \vee (x_4 < 1 \wedge x_5 < x_6)$. satisfying sets are written beside constraints as a list of values. For example, the satisfying set for c_5 is $\langle x_4, 0\rangle, \langle x_5, 0\rangle, \langle x_6, 1\rangle$. Watched literal events are omitted from this diagram.

we use for keeping track of if a constraint is satisfiable, or disentailed. The major advantage of this framework is that it allows our algorithm to use watched triggers, even when the propagators of the child constraints we are considering use backtracking or static triggers.

Definition 2. *Given a constraint C, a satisfying set is a set of literals from $\mathcal{X}_C$ such that every assignment to $\mathcal{X}_C$ which contains all the literals in F also satisfies C.*

A satisfying set is complete *if it is also satisfies the condition that every list of sub-domains of $\mathcal{X}_C$ which contains all the literals in F allows at least one assignment that satisfies C.*

Example 3. *Consider the constraint $X + Y + Z \geq 2$, for variables X, Y and Z with domains $\{0, 1\}$. The set of literals $\{\langle X, 1\rangle, \langle Y, 1\rangle\}$ is complete satisfying set. It is a satisfying set as the two assignments which contains both these literals, $\langle X, Y, Z\rangle = \langle 1, 1, 0\rangle$ or $\langle 1, 1, 1\rangle$, satisfy the constraint. It is complete because any list of sub-domains of $\mathcal{X}_C$ which contains these two assignments must contain an assignment where $X = 1$ and $Y = 1$, and therefore regardless of the assignment to Z, the sum of the variables must be greater than or equal to two.*

The set of literals $\{\langle X, 0\rangle, \langle X, 1\rangle\}$ is a satisfying set, as there is no assignment which contains two assignments to the same variable. It is however not complete, because the list of sub-domains which allow any value for X, but require 0 for both Y and Z does not contain an assignment which satisfies the constraint.

A satisfying set is simply a set of literals with particular properties. The principle behind their use is that we watch all the literals in the satisfying set, and only reinvoke the constraint if one of them is removed. As with other algorithms which use watched literals, satisfying sets perform very well when they are small, but as our experiments will show still perform well in the general case.

Given a satisfying set for a constraint, we know that if none of its literals in the satisfying set are removed, we cannot end up in a state where every variable is assigned and the constraint is not satisfied. This basic guarantee will be used to ensure algorithms using satisfying sets are correct. A complete satisfying set on the other hand produces a much stronger guarantee, that the constraint is never disentailed as long as no literal from it is lost. This will be necessary for any constraint which makes use satisfying sets to achieve GAC.

Satisfying sets do not exist for every constraint and list of sub-domains. For example, if every variable in a constraint is assigned and the constraint is not satisfied, then there is no satisfying set. A complete satisfying set cannot exist for any constraint which has become disentailed. Therefore Definition 4 introduces a satisfying set generator, which can either produce a satisfying set, or return FAIL if it deduces one does not exist. These will be used in our algorithms to efficently check if a constraint is satisfiable.

Definition 4. *A satisfying set generator for a constraint C is a function which, given a list of sub-domains, either returns a satisfying set within those sub-domains, or* FAIL *is it can prove that no the sub-domains contain no assignment which satisfies C. This implies a satisfying set generator must return* FAIL *for sub-domains where no valid*

satisfying set exists, and may return FAIL *for other domains where there is no valid solution, but valid satisfying sets exist.*

A satisfying set generator is complete *if it only returns complete satisfying sets. This implies it must return* FAIL *exactly when there is no assignment in the sub-domains which satisfies* C*.*

One obvious question is for which constraints satisfying set generators can be implemented in polynomial time, and when they can be made complete. Definition 5 presents the trivial satisfying set generator, which provides a polynomial-time satisfying set generator for any constraint. This is not the only way of constructing a generic polynomial-time satisfying set generator.

Definition 5. *The trivial satisfying set generator for a constraint* C *is defined as follows:*

1. *If the current sub-domain of any variable in* $\mathcal{X}_C$ *allows more than more value, choose one such variable and return the satisfying set containing two literals from that variable's sub-domain.*
2. *If every variable in* $\mathcal{X}_C$ *allows exactly one assignment, return the satisfying set containing those literals if that assignment satisfies* C*, else return* FAIL*.*

Lemma 6 shows that the trivial satisfying set generator is valid.

Lemma 6. *For every constraint* C*, the trivial satisfying set generator is valid and runs in polynomial time.*

Proof. The complexity result is trivial, as the algorithm requires at most checking if every variable is assigned, and then checking at most one assignment satisfies the constraint. Any set of literals which contains two assignments to one variable is a satisfying set, as no assignment to $\mathcal{X}_C$ can contain the satisfying set. Once all variables are assigned, the trivial checker either returns FAIL, or returns a complete assignment which must satisfies C. □

Complete satisfying set generators are much harder to construct. Theorem 7 shows that a constraint has a polynomial-time satisfying set generator exactly when it has a polynomial time GAC propagator.

Theorem 7. *A constraint* C *has a polynomial time complete satisfying set generator if and only if it has a polynomial time GAC propagator.*

Proof. Given a complete satisfying set generator, it is possible to check if list of subdomains for $\mathcal{X}_C$ contains a satisfying assignment, by seeing if the satisfying set generator returns FAIL. Lemma 1 of [16] proves this is polynomially equivalent to having a GAC propagator.

Alternatively, given a GAC propagator for C, we can construct a complete satisfying set generator as follows. The GAC propagator will empty the domains of the variables if the sub-domains contain no assignment which satisfies C, which is exactly the situation in which a complete satisfying set generator should return FAIL.

Assuming the GAC propagator does not empty the domains, then they must contain at least on satisfying assignment. A complete assignment which satisfies C is a valid complete satisfying set, as any sub-domain which contains it obviously contains a satisfying assignment. The following algorithm such and assignment, within $|\mathcal{X}_C|$ invocations of the propagator.

1. Run the GAC propagator.
2. If any unassigned variable exists, choose one and assign it any value.
3. If any variable is unassigned, return to step 1.

□

While Theorem 7 shows explicitly how to build a complete satisfying set generator from a GAC propagator, for the majority of constraints it is much simpler to build such a checker in practice. Further, often complete satisfying set generators will return a set of literals which contain fewer literals than the number of variables in the constraint. How to find small satisfying sets in general is an open problem. For all the constraints in Minion which have GAC propagators, it is easy to construct a complete satisfying set generator by taking a part, often but not always small and simple, of the propagator. We present a few cases here as examples.

Example 8. *Consider the constraint $\sum x_i \geq c$ for Boolean variables x_i and constant c. One complete satisfying set generator for this constraint looks for c variables which can be assigned* TRUE*, and as it soon as it finds them, returns that set of c literals. If no such set exists, return* FAIL.

Example 9. *Consider the constraint $M[x] = y$ for an array of variables M and variables x and y. Given a set of subdomains, this constraint is satisfiable if and only if there exist i and j such that i is in the domain of x and j is in the domain of both $M[i]$ and y. If such i and j exist, then the literals $x = i$, $M[i] = j$ and $y = j$ form a complete satisfying set.*

Example 10. *The complete satisfying set generator given in Theorem 7 requires finding a complete assignment which satisfies the constraint. The first part of the AllDifferent [13] propagation algorithm finds a matching which forms such a satisfying assignment, so a complete satisfying set generator can be formed by truncating the algorithm at this point.*

4 Efficient Propagators for OR, AND and ATLEASTK

In this section we present a new for the OR of a set of constraints, then show how that algorithm can be extended to ATLEASTK (where at least k of the constraints in the set are satisfied in any solution). While OR can be expressed as an ATLEASTK where $k = 1$, we shall consider OR separately, as it allows both a simpler proof of correctness and simpler algorithm. We also present a simple algorithm for AND which may be nested arbitrarily with OR and ATLEASTK.

These types of constraints occur in the models of many problems, so having an efficient implementation of them is important. Using watched literals and satisfying sets, we shall produce a very efficient implementation of these constraints, particularly large OR constraints and ATLEASTK constraints which require only a small proportion of members to be true. However, as our experiments will show, these algorithms are still useful even for short OR constraints, or ATLEASTK constraints that require a large proportion of the child constraints to be true.

4.1 Theoretical Overview

The ATLEASTK algorithm (of which OR is a special case) can be summarized as follows, for parameter k and constraint set Con. If there are more than k constraints in Con that are not disentailed, then a set of $k + 1$ of these is maintained by the algorithm. Each constraint in the set is *watched* (it is checked for disentailment) by using satisfying sets and watched literals. If there are exactly k constraints in Con that are not disentailed, they are propagated. (If there are fewer than k, the algorithm will fail.)

In Theorem 11, we prove that our algorithm enforces GAC correctly when all child constraints have a GAC propagator, and child constraints do not share variables. If the child constraints merely have correct propagators, the algorithm remains correct (although not GAC). In Section 2.1 we discuss correctness in the presence of repeated variables.

Theorem 11. *Consider a constraint C which can be expressed as "At least k of the set of constraints $Con = \{Con_1, Con_2, \ldots, Con_n\}$" for a constant k, where the scopes of the Con_i are disjoint.*

Given a non-empty sub-domain D_v for each variable in the scope of C, then the D_v are GAC with respect to C if and only if, either:

1. *At least $k + 1$ of the Con_i have satisfying assignments in the D_v*
2. *Exactly k of the Con_i has a satisfying assignment in the D_v, and the D_v are GAC with respect to each of these k constraints.*

Proof.

1. Assume that there exists some set S such that $|S| = k + 1$ and Con_i has a satisfying assignment for every $i \in S$. Then given any assignment to any variable, there are at least k members of S, which do not contain this variable in their scope. A satisfying assignment to C can be generated by assigning these k constraints a satisfying assignment, and then assigning all other variables any value. Therefore, every assignment to every variable is supported.
2. Assume there exists a set S such that $|S| = k$ and Con_i is satisfiable if and only if $i \in S$. This implies in any satisfying assignment to C, then every Con_i for $i \in S$ must be satisfied. Therefore for each $i \in S$, any assignment to any variable in the scope of Con_i which cannot be extended to a satisfying assignment to Con_i must be removed. This is the definition of $GAC(Con_i)$.

   Any variable not in the scope of any Con_i for $i \in S$ can be assigned any value.

If there are less than k members of the Con which are satisfiable, clearly no assignment can satisfy C. □

4.2 The Watched OR Propagator

Our algorithm is split into three distinct phases, namely a *setup* phase, a *watching* phase and a *propagation* phase. In this section we will present each phase separately. Before presenting the steps in our algorithm, we first describe the state that the algorithm stores between calls.

PropagateMode: a Boolean which represents if we are in the propagation phase of the algorithm. It is reverted when search backtracks.

Watches: The indices of the two child constraints that are currently being watched. These are not reverted when search backtracks.

The algorithm operates on child constraints C_1 to C_n, which are required to have a propagator and a satisfying set generator. By using the constraint trees framework (Section 3.2), the child propagators are able to use any kind of trigger available in Minion, and executing them is almost as efficient as propagating an ordinary constraint (the only overhead being passing trigger events through the OR).

The algorithm begins in the setup phase. This searches for two satisfiable children. If two can be found then they are both watched, if one is found then the propagation phase is entered, and if none are found then the constraint fails, denoting that search should backtrack.

PropagateMode = FALSE;
if $\exists i.\ C_i$ *has satisfying set* **then**
    if $\exists j.\ i \neq j \wedge C_j$ *has satisfying set* **then**
        Place watched literals on satisfying set of C_i;
        Place watched literals on satisfying set of C_j;
        Watches=$\{i, j\}$
    else
        Initialise propagation of C_i;
        PropagateMode = TRUE;
    end
else
    Fail;
end

While PropagateMode is FALSE, whenever a literal of a satisfying set is pruned, the watching phase of the algorithm is called. This either finds a new satisfying set, or (if only one child is satisfiable) starts to propagate a child.

Finally, the propagation phase is active when PropagateMode is TRUE. All trigger events belonging to C_j are passed through to C_j.

It is possible to receive stale trigger events from watched literals which were placed in a different phase because watched literals are not backtracked. Therefore in the watching and propagation phases, some trigger events must be ignored or otherwise handled specially. These are listed below.

Watching Phase: Trigger events from the propagation phase may be received in this phase; in this case the watched literal is removed and the event is ignored. Any trigger events from static triggers belonging to child constraints are ignored.

Input: i : The satisfying set of constraint C_i has been lost
Global Data: PropagateMode
if *PropagateMode* **then**
    Return
end
if C_i *is satisfiable* **then**
    Move watched literals to new satisfying set of C_i;
else
    if $\exists k.\ C_k$ *is satisfiable and* $k \notin$*Watches* **then**
        Move watched literals to satisfying set of C_k from C_i;
        Watches=$(\text{Watches} \setminus \{i\}) \cup \{k\}$;
    else
        $\{j\} = \text{Watches} \setminus \{i\}$;
        Initialise propagation of C_j;
        PropagateMode = TRUE;
    end
end

Propagation Phase: When propagating one child constraint, static trigger events for the other children are ignored. Watched literal events from setup and watching phases are ignored, and watched literal events from another child cause the corresponding watched literal to be removed.

To prove our algorithm correct, we present two invariants, which ensure our algorithm works correctly.

Lemma 12. *After the setup phase for the algorithm has completed, at any point during search where failure has not occurred and all items on the constraint queue have been executed, the following two invariants are true.*

1. *PropagateMode* = FALSE *implies that two satisfying sets of two child constraints are being watched.*
2. *PropagateMode* = TRUE *implies that* $n-1$ *child constraints are known to be unsatisfiable, and the other one is being propagated.*

Proof. The two invariants are proved separately in parts 1 and 2 below.

1. Clearly invariant 1 is true after setup, and whenever search progresses forward. However, we must consider what happens when search backtracks. If PropagateMode was TRUE and remains so, then the condition is trivially true. There are two other cases to consider.
    - Backtrack from node A where PropagateMode is FALSE to node B. At B, **PropagateMode** must still be FALSE. The two satisfying sets from A are retained, and they are valid at B since the domain sets at B are (non-strict) supersets of those at A.

- Backtrack from node A where PropagateMode is TRUE to node B where it is FALSE. The two satisfying sets were found at node B or at an intermediate state between A and B. They remain valid at B since the domain sets at B are supersets of those at any intermediate state.

2. In both places where PropagateMode is set to TRUE, the invariant holds. Suppose PropagateMode is set to TRUE at node A. For all nodes B below A in the search tree, domain sets are a subset of those at A and therefore the invariant still holds (i.e. the $n-1$ unsatisfiable children remain unsatisfiable at B). When backtracking from A, PropagateMode is reverted to FALSE therefore the invariant holds.

   Note that the propagated child may also be unsatisfiable. This can arise when the propagator does not perform GAC.

□

Our Watched OR algorithm uses satisfying sets extensively. It is interesting to also construct satisfying sets for the Watched OR constraint, so that it can be embedded in other parent constraints. Since OR is equivalent to ATLEASTK when $k = 1$, a satisfying set generator is given by Definition 13 and proof of soundness and completeness by Lemma 14 in the next section.

The algorithm presented here is a generalization of unit propagation (with watched literals) in SAT [5], although Watched OR is much more complex than unit propagation. A SAT clause is an OR of literals of Boolean variables ($\langle x_i, 0\rangle$ or $\langle x_i, 1\rangle$). Generating a satisfying set for a literal is trivial, it is simply the literal. Also, the propagation phase is trivial in SAT because a literal becomes implied after being propagated once.

4.3 The Watched ATLEASTK Propagator

ATLEASTK (defined in Section 2.1) with parameter k states that at least k constraints in a set are satisfied in any solution. This is a natural generalization of OR, which emerges when $k = 1$. We name the propagation algorithm Watched ATLEASTK.

The design of Watched ATLEASTK is similar to Watched OR. Rather than watching two disjuncts and propagating a single disjunct when all others are false, we instead watch $k+1$ disjuncts, and propagate k when all others are false. The proofs of correctness follow almost identically, as does the algorithm itself. We expect this algorithm to be most efficient when k is small compared to the size of the constraint set.

The Watched OR algorithm is adapted as follows. The Watched set of child indices now has cardinality $k + 1$ at all times, and we introduce a set named Prop containing indices of the k constraints which are propagated in the propagation phase.

The setup phase is altered to seek $k + 1$ satisfiable child constraints rather than 2. If there are exactly k satisfiable children, Prop is assigned to this set of k children, and all constraints in Prop are initialized.

In the watching phase, when a new satisfying set cannot be found, Prop is set as follows: $\mathrm{Prop} = \mathrm{Watches} \setminus \{i\}$, and all constraints in Prop are initialized, rather than only C_j.

In the propagation phase, trigger events belonging to any constraint in Prop are passed through to their owners. Therefore all constraints in Prop are propagated to a fixed point.

The proof of correctness lifts trivially to this new algorithm. In the first invariant, the number of watched constraints becomes $k + 1$. In the second invariant, the number of known unsatisfiable children becomes $n - k$ and the number of propagated children becomes k.

Definition 13. *Given satisfying set generators for a set of constraints* $\{C_1, \ldots, C_n\}$*, the satisfying set generator for* ATLEASTK $(C_1, \ldots, C_n)$ *is defined as follows:*

If the satisfying set generators of more than $n - k$ *children return* FAIL*, then return* FAIL*. Otherwise choose any set of* k *children whose satisfying set generators do not return* FAIL*, and return the union of the satisfying sets they generate.*

Lemma 14. *The satisfying set generator for* $C =$ATLEASTK $(C_1, \ldots, C_n)$ *given in Definition 13 is correct. Further, it is complete if the satisfying set generators for the* C_i *are complete and for all* $i \neq j$*,* $\mathcal{X}_{C_i}$ *and* $\mathcal{X}_{C_j}$ *are disjoint.*

Proof. (Correct) The satisfying set generator for C is correct, because a complete assignment can only contain a satisfying set it generates if it contains the satisfying sets produced by the satisfying set generators for k children, and therefore this assignment satisfies k children.

(Completeness) When the variables of all pairs of constraints are disjoint, given a list of sub-domains which contain not satisfying assignment for C there must exist at least $n - k + 1$ child constraints which are not satisfiable in these sub-domains. Therefore any satisfying set for C must incorporate a satisfying set for at least one unsatisfiable child C_i. If the satisfying set generator for C_i is complete, it will fail in this situation and the generator for C will also fail.

When C is satisfiable, complete satisfying sets may be generated for k children. By the completeness property, each satisfying set must be contained in a satisfying assignment for that child. These k satisfying sets may be joined (since the children do not share variables). Any list of sub-domains which contains the resulting satisfying set must contain an assignment which satisfies those k children, and therefore must satisfy C. Therefore the satisfying set generator for C is complete. □

While Lemma 14 shows a complete satisfying set generator for each of a set of constraints$\{C_1, \ldots, C_n\}$ leads to a complete satisfying set generator for the constraint ATLEASTK $(C_1, \ldots, C_n)$, the reverse is not true. For example, consider for example the case where k of the children are the TRUE constraint, then C is also the TRUE constraint and can always return an empty satisfying set, even if some other children have incomplete satisfying set generators. However in general the satisfying set generator for $C_1 \vee \ldots \vee C_n$ will be complete only when the satisfying set generators for all of the C_i are complete.

4.4 The Watched AND Propagator

In a similar fashion to Watched OR and Watched ATLEASTK, we can implement an AND constraint. This constraint is theoretically and practically much simpler than

the Watched OR and Watched ATLEASTK. Further, within this framework there is no advantage, either from the point of view of speed or propagation, to using a Watched AND in isolation. However, by combining Watched AND with the other constraints introduced in our framework, we will be able to provide useful practical speed-ups.

It is a well-known result [9] that given two constraints C and D with polynomial-time GAC propagators, where $\mathcal{X}_C$ and $\mathcal{X}_D$ share variables, the problem $C \wedge D$ can be an NP-hard problem. We implement propagation for $C \wedge D$ simply by using running the propagators of C and D to fixed point, which in general only achieves GAC when $\mathcal{X}_C$ and $\mathcal{X}_D$ do not overlap.

AND is equivalent to ATLEASTK when $k = n$. Therefore the satisfying set generator for ATLEASTK (Definition 13) can be used for AND, and the proof of soundness and completeness also applies (Lemma 14).

Given a constraint of the form $C_1 \wedge \ldots \wedge C_n$, it would not be worth using Watched AND, as the level of propagation our algorithm achieves is identical to imposing the C_i as seperate constraints while being slower. However, we shall show how in a constraint of the form $(C_1 \wedge D_1) \vee \ldots \vee (C_i \wedge D_i)$, for example, Watched AND forms a powerful and useful tool.

4.5 Alternatives to Parent Constraints

In Section 1.4 we discussed two alternatives to Watched OR. The first is *flattening* using reification, which is not restricted to OR, and is the most common way in which complex constraints are constructed in CP solvers. Some solvers such as Minion require the user to do this flattening, while others such as Eclipse and ILOG Solver do this flattening behind the scenes on the user's behalf. In our experiments we compare against this approach, and in all experiments it gives the same domain removals as Watched AND, OR and ATLEASTK.

The second model from Section 1.4 is specific to implementing a special case of OR. Recall that the example has two vectors X and Y, with the constraint:

$$(X[1] \neq Y[1]) \vee \cdots \vee (X[n] \neq Y[n])$$

There are two problems which arise when this disjunction is implemented using the three constraints $X[i] = x, Y[i] = y, x \neq y$ and auxiliary variables i, x and y. Consider the following domain sets:

$$\begin{array}{lll} X[0] \in \{0\} & X[1] \in \{1\} & \\ Y[0] \in \{1\} & Y[1] \in \{0\} & \\ i \in \{0,1\} & x \in \{0,1\} & y \in \{0,1\} \end{array}$$

Propagating each of the three constraints $X[i] = x, Y[i] = y$ and $x \neq y$ in isolation on these domains does not lead to any domain reductions. Therefore not only does this representation of the disjunction not achieve GAC, but even after assigning all variables in the arrays X and Y to an assignment which does not satisfy the constraint failure does not occur. It is necessary to also assign the new auxiliary variables before one of the constraints fail, and search backtracks.

The second problem comes from the fact that there may be many index values i where $X[i] \neq Y[i]$. In this case, a different solution will be generated for each possible assignment to the index variable. For a problem with many disjunctive constraints, this can lead to each original solution resulting in an exponential number of solutions.

These problems lead to this representation performing very poorly in practice, as we show in Section 4.6.1.

4.6 Experimental Results

We claimed in Section 1.2 that both constraint trees and movable triggers are essential for propagation of logical connectives. Here we test that claim on four different problems.

Unless otherwise stated, we give node counts as reported by the Minion solver, and times on a 1.6 GHz Intel Core Duo with 2 GB RAM, running Mac OS X.

4.6.1 The Generalised Pigeon-Hole Problem

The first experiment is a generalisation of the pigeon-hole problem. Rather then the traditional problem of finding assignments to an array of variables which are all different, we instead consider the problem of finding assignments to a two-dimensional array of variables, where each row must be different.

The parameters are the number of rows n, the length of rows p, and the domain size d. All models have $n(n-1)/2$ not-equal constraints between pairs of rows. We compare five representations of the not-equal constraint.

Watched OR: Implemented as a Watched OR, the algorithm described in Section 4.2.

Element: The second model from Section 1.4.

Sum: The first model from Section 1.4.

Watched Sum: The same model as **sum**, except the sum constraint is replaced by a watched SAT clause $b[1] \vee \ldots \vee b[l]$.

Custom: A custom-written propagation algorithm using static assignment triggers on all variables, enforcing the same level of consistency as Watched OR (GAC).

We explore all four possibilities of using static or movable triggers, with reification or constraint trees, as shown in the table below.

	Static Triggers	Movable Triggers
Reification	**Sum**	**Watched Sum**
Constraint Trees	**Custom**	**Watched OR**

Note that using Theorem 6.6 from [17], as long as we get GAC on each of the constraints in the **Sum** and **Watched Sum** models, we get GAC over the whole OR, and further as long as we place the new variables at the end of the search ordering, the resulting searches will be identical to the Watched OR model. Therefore, the only model which could result in a different sized search is **Element**.

$\langle n, p, d \rangle$	Element		Watched OR	
	Time	Nodes	Time	Nodes
$\langle 8, 3, 2 \rangle$	27	12,335,593	0.05	25
$\langle 8, 3, 3 \rangle$	6,245	3,112,501,760	0.09	28
$\langle 8, 4, 2 \rangle$	2,223	1,092,789,218	0.05	33
$\langle 8, 4, 3 \rangle$	>100,000	>45,000,000,000	0.11	26

Table 1: Search size for small instances of the array pigeonhole problem

Length p	Domain d	**Watched OR**	**Sum**	**Watched Sum**	**Custom**
5	2	313,459	38,577	51,758	74,389
10	2	989,251	3,085	3,149	111,947
20	2	4,142,598	989	1,000	85,723
30	2	4,176,330	630	630	78,276
40	2	4,334,456	465	472	96,441
50	2	3,964,028	374	377	66,531
5	10	1,939,067	8,230	8,227	87,851
10	10	1,502,164	3,373	3,470	48,434
20	10	464,445	997	1,004	60,249
30	10	281,841	615	616	57,474
40	10	210,891	455	458	46,929
50	10	176,598	365	366	43,433

Table 2: Nodes per second averaged over 100 seconds of pigeonhole instances where $n = 100$

Since we achieve GAC, there is no scope for Lhomme's algorithm [10, 11] (or other constructive disjunction algorithms) to enforce a stronger consistency. Lhomme's algorithm is statically triggered, and would be similar to **Custom** in this context.

Table 1 shows just how badly the **Element** model performs in practice on some very small instances, quickly leading to insolvable problems which the other models we consider are all able to solve in less than a second. Due to to the very poor performance of this model, it will not be considered further. As the remaining four models produce identical search trees, we shall only compare them in terms of the number of nodes of search they perform per second.

In Table 2, we compare the node rate on various instances of the generalised pigeonhole problem. The **Custom** model improves significantly on **Sum** and **Watched Sum** by eliminating the additional variables, but Watched OR is always faster than **Custom**, sometimes by several orders of magnitude. When in the watching phase, the Watched OR algorithm will use only four watched literals: two for each watched child constraint. By comparison, the custom algorithm has assignment triggers on all variables. This illustrates the importance of using an appropriate triggering mechanism, in this case watched literals.

With domain size 2, the Watched OR algorithm sometimes increases in speed as instance size increases. This surprising result is caused by a decrease in the proportion

of variables with a watched literal on them.

The small differences between **Sum** and **Watched Sum** show that the gain from using watched literals for the sum constraint is often insignificant compared to the cost of propagating the reified not-equal constraints.

In summary, these results support the hypothesis that both constraint trees and movable triggers are required to efficiently propagate OR.

4.6.2 The Anti-Chain Problem

In our second experiment we consider the anti-chain problem, defined below.

Definition 15. *An **anti-chain** is a set S of multisets where $\forall\{x, y\} \subseteq S.\ x \nsubseteq y \wedge y \nsubseteq x$.*

This is modelled as a CSP as follows. The $\langle n, l, d\rangle$ instance of anti-chain is a CSP with n arrays of variables, denoted $M_1, \ldots, M_n$, each containing l variables with domain $\{0, \ldots, d-1\}$ and the constraints $\forall i \neq j \in \{1, \ldots, n\}.\ \exists k \in \{1, \ldots, n\}.\ M_i[k] < M_j[k]$.

Each variable $M_i[v]$ represents the number of occurrences of value v in multiset i, up to a maximum of $d-1$. Each pair of rows M_i and M_j differ in at least two places: in one position k, $M_i[k] < M_j[k]$ and in another position p, $M_i[p] > M_j[p]$. This ensures that neither multiset contains the other.

Similarly to the generalised pigeon-hole problem, we consider 4 implementations of the constraint $\exists i.\ M[i] < N[i]$ for arrays M and N.

Watched OR: Implemented as a Watched OR.

Element: Introduce variables i with domain $\{0, \ldots, l-1\}$ and m and n each with domain $\{0, \ldots, d-1\}$. Impose the three constraints $M[i] = m, N[i] = n$ and $m < n$.

Sum: Introduce a new array of Boolean variables b of length l and impose the set of constraints $\forall i.\ (M[i] < N[i]) \leftrightarrow b[i]$ and also $\sum(b_{ij}) \geq 1$.

Watched Sum: The same algorithm as **sum**, except the constraint $\sum(b \geq 1$ is replaced by a watched SAT clause $b[1] \vee \ldots \vee b[l]$.

We did not construct a custom propagator for this experiment because it takes considerable effort and we are concerned with generic algorithms.

Similarly to the previous experiment, the Watched OR, **Sum** and **Watched Sum** all enforce the equivalent of GAC on the original expression, and **Element** does not.

Once again, we will consider the **element** model separately, as we must compare time, rather then just nodes per second. In each of these experiments, we search for only the first solution and results are given in Table 3.

These results are much closer than those in the pigeon hole problem. On some instances, such as $\langle 11, 5, 2\rangle$, the **Element** model even achieves the same sized search as Watched OR. However, **Element** was slower in terms of nodes per second on all the instances we considered. Furthermore, **Element** sometimes exhibits a much larger number of solutions. Table 4 shows the results of finding all solutions to a small set of

$\langle n, l, d \rangle$	**Element**		**Watched Or**	
	Time	Nodes	Time	Nodes
$\langle 11, 4, 3 \rangle$	3.3	142,674	0.3	77,177
$\langle 12, 4, 3 \rangle$	68	3,030,555	6.5	2,189,034
$\langle 13, 4, 3 \rangle$	3,812	166,888,355	286	95,301,659
$\langle 14, 4, 3 \rangle$	4158	166,888,372	301	95,301,661
$\langle 9, 4, 10 \rangle$	3.3	90,678	0.14	12,349
$\langle 10, 4, 10 \rangle$	25.5	636,635	0.4	75,807
$\langle 11, 4, 10 \rangle$	153	3,340,225	2.1	399,997
$\langle 12, 4, 10 \rangle$	681	3,340,255	11	1,815,755
$\langle 11, 5, 2 \rangle$	29.34	2,411,733	7.91	2,411,733

Table 3: Search size for Finding the first solution to the antichain problem

$\langle n, l, d \rangle$	**Element**			**Watched Or**		
	Time	Nodes	Solutions	Time	Nodes	Solutions
$\langle 2, 4, 3 \rangle$	0.1	18,628	8,748	0.05	8,099	4,050
$\langle 3, 4, 3 \rangle$	3.2	1,269,108	2,855,281	0.1	288,377	144,150
$\langle 4, 4, 3 \rangle$	665	561,666,863	240,375,312	3.6	7,657,223	3,823,200
$\langle 3, 6, 2 \rangle$	2.3	3,102,719	1,551,360	0.1	167,999	84,000
$\langle 3, 7, 2 \rangle$	50	70,533,119	35,266,560	0.8	1,845,143	922,572

Table 4: Finding all solutions for instances of the antichain problem

problems. The number of solutions found by the **Watched Or** model is the correct number of solutions, the **Element** duplicates some of these solutions multiple times, due to the fact its auxiliary variables can take multiple values for each solution to the problem. This shows once again the limitation of the **Element** model in practice.

To compare the other three models we consider how many nodes per second the particular model can solve, averaged over the first 100 seconds of search. In both cases we consider solving the anti-chain problem on 100 arrays ($n = 100$) of varying length and domain size.

A number of conclusions can be drawn from the results of this experiment, given in Table 5. First of all, our algorithm performs well compared to **Sum** on short vectors, but improves steadily as the length increases. For example with Boolean domains for length 5 arrays our algorithm is around 15 times faster, improving to 18 times for length 50. We note that for larger domains the nodes per second increases as the problem size increases. This is in common with the pigeon-hole problem, and is caused by a decrease in the proportion of variables with a watched literal on them.

This experiment partially supports the hypothesis that both constraint trees and movable triggers are required to efficiently propagate Or. However we do not have an algorithm using static triggers with constraint trees, so we have not fully explored the space.

Length l	Domain d	**Watched Or**	**Sum**	**Watched Sum**
5	2	47,970	3,285	3,137
10	2	38,316	2,793	2,402
20	2	28,550	2,028	1,771
30	2	21,858	1,627	1,288
40	2	18,250	1,226	979
50	2	15,820	888	716
5	10	1,467	77	70
10	10	1,228	71	64
20	10	1,330	70	65
30	10	1,487	69	65
40	10	1,971	72	69
50	10	2,249	76	72

Table 5: Nodes per second achieved on antichain instances

4.6.3 The Hamming Codes Problem

The final problem considered is Hamming codes, defined below.

Definition 16. *The $\langle n, l, d, s\rangle$ instance of the Hamming problem is to find a set of n codewords of length l with alphabet $\{1 \dots d\}$, where each pair of codewords differ in at least s positions.*

This is modelled as follows. We have n arrays of integers, named $M_1, \dots, M_n$, each of length l and domain $\{1, \dots, d\}$ with the following Hamming distance constraints: $\forall \{i,j\} \subseteq \{1, \dots, n\}.\ \left(\sum_{k \in \{1,\dots,l\}} M_i[k] \neq M_j[k]\right) \geq s$. We compare 3 representations of the constraint $(\sum_{i \in \{1,\dots,l\}} M[i] \neq N[i]) \geq s$.

Watched AtLeastK: Directly represented as a Watched AtLeastK, the algorithm described in Section 4.3.

Sum: Introduce an array of auxiliary Boolean variables $b[l]$ and add the set of constraints $\forall i \in \{1, \dots, l\}.\ (M[i] \neq N[i]) \leftrightarrow b[i]$. Then impose $\sum b \geq s$.

Watched Sum: The same model as **Sum**, except the constraint $\sum b \geq s$ is replaced by a watched sum constraint.

For this problem we do not attempt to give an **Element** model, because preliminary experiments showed that the performance was so poor it was impossible to usefully compare it to any of the other models.

We experimented with the Hamming codes problem where $n = l = 50$ and $d = 2$, and the Hamming distance s is varied. The results are presented in Table 6. As stated in Section 4.3, we expect the Watched AtLeastK algorithm to be most efficient when k is small (where $k = s$ here). This is supported by Table 6, which shows Watched AtLeastK performing much better at low values of s than high values. Watched AtLeastK dominates **Sum** and **Watched Sum** when $s \leq 45$. When $s = 49$, **Sum**

Distance s	**Watched AtLeastK**	**Sum**	**Watched Sum**
49	29,879	37,425	13,494
45	80,496	72,225	19,625
40	88,030	83,900	27,651
30	159,411	95,964	41,254
20	175,656	99,757	59,059
10	90,787	29,743	19,923
5	3,710,787	2,616	2,598
3	4,821,390	2,146	2,175
2	4,848,664	2,089	2,092

Table 6: Nodes per second for Hamming instances where $n = 50$, $l = 50$ and $d = 2$

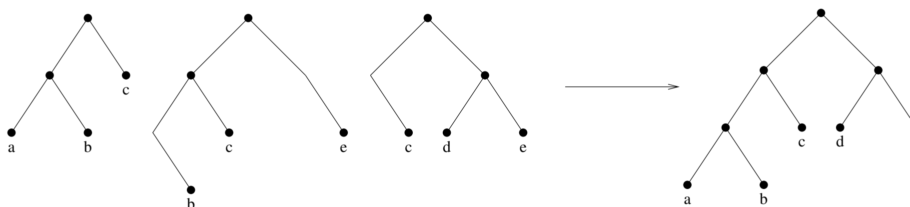


Figure 2: A toy input (left) and single solution to the supertree problem (right). Input trees are distorted to make relationships in resultant supertree more obvious.

performs best. In this case, Watched AtLeastK will watch all child constraints, so there is little scope for it to improve on **Sum**. However Watched AtLeastK does remain competitive.

In summary, this experiment provides some evidence that the gains from using constraint trees and movable triggers apply to AtLeastK as well as Or.

4.6.4 The Supertree Problem

The supertree problem [18] is that of transforming an input set of rooted bifurcating trees (*species trees*), describing the evolutionary history of a set of species, into an output tree respecting all the relationships in the input. That is, if species a is more closely related to b than to c in the input, then it must be so in the output. An example is shown in Figure 2. Notice, for example, that in the input a and b are more closely related to each other than to c (their common ancestor is deeper in the tree), and this property is maintained in the output.

Various CP models have been created to solve this problem, here we will use the model of [19] as well as the optimisation model of [20]. Both consist almost entirely of constraints of the form $(a \leq b = c) \vee (b \leq a = c) \vee (c \leq a = b) \vee (a = b = c)$. Each one ensures that the set $\{a, b, c\}$ of species must have a correct evolutionary relationship, ie., one pair must be most closely related, or all 3 are equally related. The standard model requires all such constraints to be satisfied, while the optimisation model maximises the number that are satisfied.

Instance	Nodes	Watched solve time	**Sum** solve time	Saving
AB	58	0.154976	0.191971	19%
AD	97	0.227965	0.395940	42%
AF	66	0.160975	0.204969	21%
AG	152	0.209968	0.378943	45%
BD	72	0.263959	0.350946	25%
BF	27	0.156976	0.196970	20%
BG	78	0.204969	0.299954	32%
CD	53	0.251962	0.453931	44%
CF	30	0.172974	0.231965	25%
DF	81	0.377942	0.442932	15%

Table 7: Experimental data for solvable supertree instances

This can be modelled directly as $\text{OR}(\text{AND}(a \leq b, b = c), \text{AND}(b \leq a, a = c), \text{AND}(c \leq a, a = b), \text{AND}(a = b, a = c, b = c))$ using Watched AND and Watched OR. Note that this modelling does not require any auxiliary variables. The conjuncts and disjuncts share variables, so GAC may not be enforced by the Watched AND and OR propagators.

We compare this to the **Sum** model. We have already described how OR is handled using sums (Section 1.4). To represent AND, we reify each conjunct, and then use a sum constraint to represent the conjunction. For example, we encode $\text{AND}(a \leq b, b = c)$ as $r_1 \leftrightarrow a \leq b; r_2 \leftrightarrow b = c; r \leftrightarrow (r_1 + r_2 \geq 2)$. The variable r now represents the truth of the AND constraint. This encoding uses auxiliary variables and enforces *the same* level of consistency as the above Watched OR and AND encoding.

We use all instances from Moore and Prosser [20] that have two input trees and are small enough to load. (The model takes cubic space and the larger instances exceeded 2GB RAM.) These are partitioned into ten solvable instances and four instances where input trees contain conflicting information (e.g. tree 1 says that a and b are closer relatives to each other than to c, whereas tree 2 says that a and c are closest). The standard model is used for the solvable instances, and the optimization model for the unsolvable ones. The experiment was performed on an Intel Core 2 Duo T7100 1.80GHz.

Table 7 shows that the watched model is significantly faster than **Sum** for the ten solvable instances. These times do not include time to load instances, however load times are larger for **Sum** because it is less concise. Table 8 presents results for the unsolvable instances. We ran these instances to 2,000,000 nodes and again the results are in favour of the watched model. Using a profiler we discovered that the speedups are due to an increase in propagation speed; the reduced burden of auxiliary variables has an insignificant effect in this case.

In summary, this final experiment provides some evidence that the Watched OR algorithm is valuable when combined with another parent constraint.

Instance	Best solution found	Watched solve time	**Sum** solve time	Saving
AC	48 cons satisfied	417.881473	2907.842941	86%
BC	27	576.805313	5642.960139	90%
CG	13	160.855546	288.511140	44%
DG	43	481.174850	1975.158730	76%

Table 8: Experimental data for supertree optimisation instances

5 Reification

The *reification* of a constraint C produces another constraint C_r, such that C_r has an extra Boolean variable r in its scope, and (in any solution) r is set to true iff the original constraint C is satisfied.

$$C_r \stackrel{def}{\equiv} r \Leftrightarrow C$$

Reification is a standard technique to increase the flexibility of constraint programming. Constraints can be combined in arbitrary ways using reification. For example, consider the exclusive-or of a set of constraints, as follows.

$$C_1 \oplus C_2 \oplus \ldots \oplus C_n$$

An odd number of these constraints must be satisfied in any solution. We could find no compact encoding of this structure into OR, AND or ATLEASTK. (It could be encoded as an OR of AND, where each AND states that an odd-sized subset of the constraints $C_1 \ldots C_n$ are satisfied, and others are not. However, there are exponentially many such subsets.) It is straightforward to represent this structure with reification. The constraints $C_1 \ldots C_n$ are each reified, creating extra variables $r_1 \ldots r_n$. These are added using a sum constraint, and the total variable is constrained to be odd.

In this section we describe two ways to propagate reified constraints, and compare them empirically. The first method uses only static triggers. The second method uses watched literals, and is more complex, but it overcomes some of the apparent disadvantages of the first method.

We also investigate another form of reification, which we call *reifyimply*, where the reification variable implies the constraint, as follows.

$$C_{ri} \stackrel{def}{\equiv} r \Rightarrow C$$

Again we describe an algorithm based on checking and a watched literal algorithm to propagate reifyimplied constraints.

5.1 Theoretical Analysis

Theorem 17 provides a simple algorithm which achieves GAC propagation for $r \Leftrightarrow C$, given a GAC propagator for both C and $\neg C$. We shall consider two different ways of making this algorithm more efficient, using incrementality. In general the propagators

for C and $\neg C$ will be very different and can have very different complexities. Lemma 18 shows that the propagator for $r \Leftrightarrow C$ is tractable if and only if the propagators for both C and $\neg C$ are tractable.

Theorem 17. *The following algorithm is a GAC propagation algorithm for $r \Leftrightarrow C$ for boolean variable r and any constraint C, assuming r is not in the scope of C and that the propagators for C and $\neg C$ achieve GAC propagation.*

Input*: r, C: subdomains*
if $Domain(r) = \{\text{TRUE}, \text{FALSE}\}$ **then**
  if *There is no satisfying assignment to* C **then**
    $r \neq \text{TRUE}$
  end
  if *There is no satisfying assignment to* $\neg C$ **then**
    $r \neq \text{FALSE}$
  end
end
if $Domain(r) = \{\text{TRUE}\}$ **then**
  Propagate(C)
else
  if $Domain(r) = \{\text{FALSE}\}$ **then**
    Propagate$(\neg C)$
  end
end

Proof. Consider the following cases upon entering the algorithm:

1. $Domain(r) = \{\textbf{TRUE}, \textbf{FALSE}\}$**:** In this case, we check if the values in r are supported. This requires finding both an assignment to $scope(C)$ which satisfies C, and an assignment which does not satisfy C. If either value is unsupported it is removed, and the algorithm continues with case 2 below.

   If neither value of r is removed then every value in the domain of every variable in the scope of C is supported, by either C or $\neg C$. Any assignment to the variables in C can be extended to a satisfying assignment to $r \Leftrightarrow C$ by adding either $r = \text{TRUE}$ or $r = \text{FALSE}$, depending on whether the assignment satisfies C or $\neg C$.

2. $Domain(r)$ **contains a single value:** In this case, if the domain of r is $\{\text{TRUE}\}$, $r \Leftrightarrow C$ is exactly equivalent to C, and if the domain of r is $\{\text{FALSE}\}$, the constraint is equivalent to $\neg C$.

□

Lemma 18. *GAC$(r \Leftrightarrow C)$ is NP-hard if and only if at least one of GAC(C) and GAC$(\neg C)$ is.*

Proof. Theorem 17 demonstrates how to implement GAC$(r \Leftrightarrow C)$ using at most one invocation of GAC(C) and GAC$(\neg C)$, in a polynomial-time algorithm. Therefore

GAC($r \Leftrightarrow C$) is polynomial time if both GAC(C) and GAC($\neg C$) are. By assigning r to TRUE or FALSE, we can see that GAC($r \Leftrightarrow C$) must be at least as hard as both GAC(C) and GAC($\neg C$). □

Theorem 19 presents a basic algorithm for implementing the constraint $r \Rightarrow C$. We will improve this basic algorithm using incrementality.

Theorem 19. *The following algorithm is a GAC propagation algorithm for $r \Rightarrow C$ for boolean variable r and any constraint C, assuming r is not in the scope of C and the propagator for C achieves GAC.*

```
Input: r, C: subdomains
if Domain(r) = {TRUE, FALSE} then
    if There is no satisfying assignment to C then
        r ≠ TRUE;
    end
else
    if Domain(r) = {TRUE} then
        Propagate(C)
    end
end
```

Proof. This proof follows the cases in the algorithm:

1. $Domain(r) = \{$**TRUE**, **FALSE**$\}$**:** In this case, every value in the domain of every variable in the scope of C is supported, as an assignment which contains $r =$ FALSE satisfies the constraint. Therefore the only value which could possibly be eliminated is $r =$ TRUE. This value is allowed if and only if there exists an assignment to $scope(C)$ which satisfies C.

2. $Domain(r)$ **contains a single value:** In this case, if the domain of r is $\{$TRUE$\}$, the constraint is exactly equivalent to just C, and if the domain of r is $\{$FALSE$\}$, any assignment satisfies the constraint so no pruning can occur.

□

5.2 Algorithms for Reification and Reifyimply

The following algorithms for $r \Leftrightarrow C$ and $r \Rightarrow C$ have some features in common. They all have a phase for checking entailment/disentailment of C, so that r can be set when necessary (the *watching* or *checking* phase). They all have a phase for propagating C (or $\neg C$) when that is necessary (the *propagation* phase). The watched literal algorithms also have a setup phase where watched literals are placed for the first time.

All constraints in the Minion solver have a method for performing the initial propagation of the constraint (named fullPropagate), which must be called before passing trigger events to the constraint. Each of the algorithms below has one item of backtracking state: a Boolean named FULLPROPAGATECALLED which tracks whether C (or $\neg C$) has been initialised. When the reification algorithm begins to propagate

C or $\neg C$, it calls fullPropagate[2] and sets FULLPROPAGATECALLED to true. Until FULLPROPAGATECALLED is reverted, it is safe to pass trigger events to the constraint. FULLPROPAGATECALLED also indicates when the algorithms are in the propagation phase.

When describing the algorithms, C is described as a *child* constraint object, with methods for propagation, checking disentailment (*checkUnsat*) and a satisfying set generator. Checking for disentailment is very similar to a satisfying set generator, except it does not return a satisfying set, only if the satisfying set generator would return FAIL or not. This means it can often be implemented more efficiently. Full reification also has $\neg C$ as a child. Child constraints do not receive trigger events unless they are passed through by the parent.

5.3 Static Reification

First we will describe an algorithm based on checking, which we will call *static* reification. The term static refers to the type of triggers which are used. It requires both the positive and negative child constraints to have the checkUnsat method which checks if the constraint is *disentailed*, i.e. no assignment of its variables will satisfy the constraint. Before search begins, the (static) triggers of both the positive and negative constraints are placed on the variables, along with a trigger on the reification variable.

During search, the algorithm has two phases. The first, which is active while r is unassigned, checks both the positive and negative constraints for disentailment. Whenever a trigger event is received, the owner of the trigger (the positive or negative child constraint) is determined and checkUnsat is called on that child constraint. If a child is disentailed, then r is assigned to the appropriate value, and the algorithm enters the second phase.

The second phase is active when r is assigned. This phase is entered when r is set, and exited on backtracking. In this stage, one of the child constraints is propagated, and the other is ignored, according to the value of r. The reify constraint will continue to receive trigger events for both child constraints, and one set of trigger events will be passed through and the other set ignored.

While this algorithm is simple and quick, there are a number of disadvantages. In the checking stage, all the triggers of both child constraints are used, and all trigger events from both sets of triggers cause a check to be carried out. In the next section we describe how the number of checks can be reduced using watched literals. The second disadvantage is in the second stage of the algorithm. The triggers for both child constraints are present, therefore the reifier is needlessly notified of both sets of events. Thirdly, it is not possible to embed a watched literal constraint in this reifier, because the checking stage requires a set of static triggers. Fourthly, this algorithm will propagate a child constraint when the child is entailed. This occurs whenever the first stage determines that one child is disentailed; the other must be entailed, and it is propagated needlessly in the propagation phase.

This algorithm will achieve GAC iff both the child constraints achieve GAC, and

[2] fullPropagate cannot be called when the variable event queues are not empty. To avoid this problem, the reified constraint is delayed until the variable event queues are empty.

both checkUnsat methods are exact (i.e. they both return true as soon as the constraint is disentailed).

5.4 Watched Reification

As an alternative to static reification, we propose a scheme based on watched literals. In this scheme, both the positive and negative child constraints must implement a satisfying set generator. Watched reification has three phases, described below. There are four sets of triggers: static triggers required by the child constraints; the static trigger on r; watched literals placed in phases 1 and 2 to watch satisfying sets; and watched literals placed in phase 3 by child constraints. Unlike static reification, the child constraints are allowed both static triggers and watched literals.

Setup Phase: Place static triggers for both child constraints, and on r. If r is assigned, move to the propagation phase. Otherwise, call the satisfying set generator for both child constraints. If either child returns FAIL, then it is disentailed. Set r appropriately and move to the propagation phase. Otherwise, place watched literals on both satisfying sets and move to the watching phase.

Watching Phase: If r is assigned, move to the propagation phase. If a domain value being watched is removed, then determine which child it belongs to, and call the satisfying set generator again for the child. If it returns FAIL, set r appropriately and move to the propagation phase. If it returns a satisfying set, place watched literals on it and remain in this phase.

Propagation Phase: If $r = 1$ then propagate the positive constraint, otherwise propagate the negative constraint. Trigger events for the appropriate child constraint are passed through.

Since watched literals are not backtracked, it is possible to receive stale trigger events from watched literals which were placed in a different phase. Therefore in the watching and propagation phases, some trigger events must be ignored or otherwise handled specially. These are listed below.

Watching Phase: Trigger events from the propagation phase may be received in this phase; in this case the watched literal is removed and the event is ignored. Any trigger events from static triggers belonging to child constraints are ignored.

Propagation Phase: When propagating one child constraint, static trigger events for the other child are ignored. Watched literal events from setup and watching phases are ignored, and watched literal events from the other child cause the corresponding watched literal to be removed.

Notice that watched literals from the setup and watching phases are not removed in the propagation phase. When backtracking into the watching phase, there is no opportunity to place watched literals, so the previous set must still be present.

The setup phase only occurs at the root node of search. The other two phases occur during search, and FULLPROPAGATECALLED indicates which phase the algorithm is in.

This algorithm solves the first three of the problems noted for the static reification, however it has additional costs. Calling the satisfying set generator is more expensive than calling the disentailment checker, and in this algorithm watched literals are moved and removed. Watched literal operations are $O(1)$ but they may still represent a significant overhead.

In common with static reification, this algorithm propagates entailed child constraints. We leave this for future work.

5.5 Static Reifyimply

The static reifyimply algorithm is similar to the static reification algorithm. It is simpler because it is never necessary to propagate the negation of the child constraint C, or check if the negation of C is disentailed. Therefore, this algorithm has only one child constraint. Before search begins, the static triggers of the child constraint are placed on the variables, along with a trigger on the lower bound of the reification variable r. The trigger on r generates an event only when r is set to 1.

During search, the algorithm has two phases. The checking phase, which is active while r is unassigned, checks the child constraint for disentailment. If a trigger event is received for r, then r must be set to 1, and the algorithm moves into its second phase. For any other trigger event, the algorithm first checks if r is assigned to 0. If it is, then the algorithm returns immediately. Otherwise the algorithm calls checkUnsat on the child constraint. If the child is disentailed, then r is set to 0.

The propagation phase is active whenever $r = 1$. This phase is entered when r is set to 1, and exited on backtracking. The child constraint is propagated, with all trigger events received by reifyimply being passed through to it. The program variable FULLPROPAGATECALLED is true when the algorithm is in the propagation phase.

This algorithm will achieve GAC iff the child constraint propagator achieves GAC, and its checker is exact.

5.6 Watched Reifyimply

As an alternative to static reifyimply, we propose an algorithm which makes use of watched literals in the same way as watched reification. The child constraint must implement a satisfying set generator. Watched reifyimply has three phases, described below. There are four sets of triggers: static triggers which the child constraint requires; the static trigger on r; watched literals placed in phases 1 and 2 to watch satisfying sets; and watched literals placed in phase 3 by the child constraint. Unlike static reifyimply, the child constraint is allowed both static triggers and watched literals.

Setup Phase: Place static triggers for the child constraint, and a static lowerbound trigger on r. If r is assigned to 1, move to the propagation phase. If r is assigned to 0 then return. Otherwise execute the satisfying set generator for the child constraint. If it returns FAIL, then set r to 0 and return, otherwise place watched literals on the satisfying set and move to the watching phase.

Watching Phase: If r is assigned to 1, move to the propagation phase. If r is assigned to 0 then return. Otherwise, when a domain value being watched is removed,

call the satisfying set generator again for the child constraint. If it returns FAIL, set r to 0 and return. If it returns a satisfying set, place watched literals on it and remain in this phase.

Propagation Phase: This phase is entered when r is set to 1 and exited on backtracking. The child constraint is propagated. All trigger events for the child constraint are passed through.

In common with watched reification, it is possible to receive stale trigger events from watched literals which were placed in a different phase because watched literals are not backtracked. Therefore in the watching and propagation phases, some trigger events must be ignored or otherwise handled specially. These are listed below.

Watching Phase: Trigger events from the propagation phase may be received in this phase; in this case the watched literal is removed and the event is ignored. Any trigger events from static triggers belonging to the child constraint are ignored.

Propagation Phase: Watched literal events from the setup and watching phases are ignored.

Notice that watched literals from phases 1 and 2 are not removed in phase 3. When backtracking into the watching phase, there is no opportunity to place watched literals, so the previous set must still be present.

5.7 Empirical comparison of reification algorithms

In the section we give an empirical comparison of reify and reifyimply, in their watched and static forms, using a range of realistic benchmark problems.

Notice that checkUnsat (CU) in static reification, and satisfying set generators (SSG) in watched reification perform similar tasks. Both determine whether a constraint is disentailed. Satisfying set generators additionally return a satisfying set of literals when the constraint is not disentailed. For all reified or reifyimplied constraints in the benchmarks, the two functions are equivalent for determining disentailment. Hence, static and watched algorithms provide the same level of consistency, and the solver explores the same number of search nodes for all benchmarks.

One metric we use to compare static and watched algorithms is the number of calls made to CU and SSG. Consider a hypothetical solver which only offers triggers (static or watched) on individual domain values. CU must have static triggers on any value which may be important at any time during search. SSG is able to place watches during search. In this solver SSG cannot be called more times than CU. In most cases, this carries through to Minion, however Minion has assignment triggers which are not available to SSG. For watched literals to have any potential, the number of calls to SSG must be substantially fewer, since the cost of calling it is somewhat higher and there is the additional overhead of placing watched literals.

Benchmarks are available at http://minion.sourceforge.net/benchmarks.html. The experiments in Sections 5.7.2 and 5.7.3 use the median of five runs on a Intel Core 2 Duo T7100 1.80GHz. The experiments in Sections 5.7.1 and 5.7.4 use the median of three runs on a Intel Xeon E5430 2.66GHz CPU.

Instance	Watched time (s)	Calls to SSG	Static time	Calls to CU
40	1202	151	1243	1350922599
50	1150	131	1276	758088075
60	1237	156	1287	1193581857
70	1487	256	1520	1589486539
80	1727	282	1750	2525193986
90	2022	320	2062	2596608279

Table 9: Times and call counts for steelmill problems

5.7.1 Steel Mill Slab Design

Our first benchmark consists of instances of the steel mill slab design problem [21]. This is a well-known optimisation problem involving assigning orders to a steel mill to slabs, minimising the total waste. Our instances include reifyimplied lex ordering constraints on rows of a 0/1 matrix, these constraints break symmetry on the rows and are reifyimplied so that they can be switched off when a row (corresponding to a slab) is not needed to fulfil the set of orders.

Our evaluation on this instance exhibits solid results in favour of watched reifyimply. Table 9 shows an exceptional decrease in calls to SSG compared to CU, for watched versus static reifyimply, running the instances up to 100,000,000 nodes. Here billions of calls are being made to CU compared to hundreds for SSG. In fact, after the first 100 nodes of search in all these examples, the watched assignment is hardly ever disturbed. Instance 90 is typical: during the first 100 nodes, SSG is called 260 times; at 10,000 nodes it has been called 301 times; and at 1,000,000 nodes it has been called 315 times. For the same instance CU is being called over 60 times per node on average up to 1,000,000 nodes. This dramatic improvement is due to the watched assignment being very stable in the watched variant, whereas for the static variant the bound triggers are being woken up frequently even when the constraint remains satisfiable. SSG needs to watch just two values in the scope of the lexleq needed to ensure it remains satisfiable, whereas CU has bound triggers on all the variables in the scope of the constraint.

Table 9 shows that this improvement in calls translates to an improvement in solution time. This improvement is relatively small in absolute terms, but this is because most of the time is spent propagating other constraints besides reifyimply.

With the aid of a profiler, we have discovered that, on benchmark 90, the average call to SSG for the lexicographic ordering constraint consumes 2695 CPU instructions whereas the average call to CU consumes just 54. These statistics give an impression that the SSG watches must be disturbed substantially less often than the static triggers to justify the cost, in this case more than 50 times less often (since there is an additional overhead of placing dynamic triggers on the literals).

5.7.2 Blackhole Solitaire

Blackhole solitaire [22] is a single-player card game. The initial layout is 17 stacks of 3 cards, with all cards visible. There is one special stack, containing only the ace of

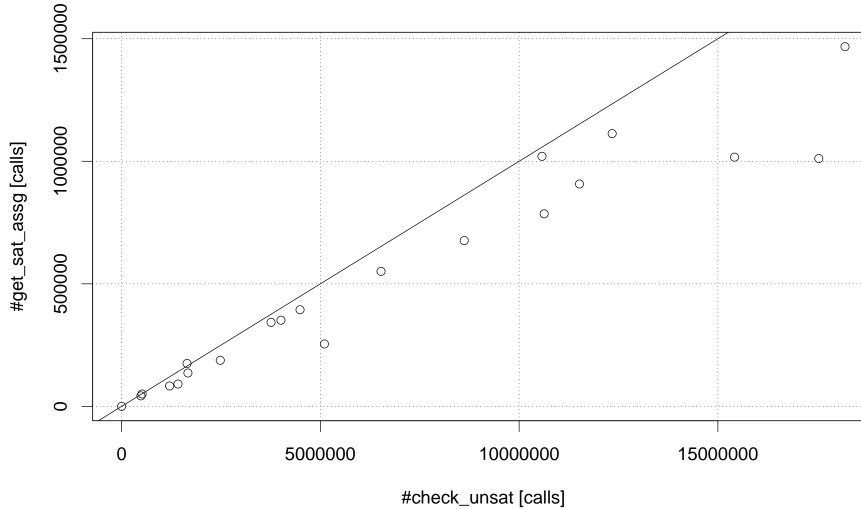


Figure 3: Comparison of calls to SSG and CU for blackhole problems

spades initially, named the black hole. Cards are moved from the top of a stack onto the black hole, and the game is completed when all 51 cards have been moved onto the black hole. The card moved must be adjacent to (but not the same as) the previous card on the black hole, regardless of suit, where adjacency wraps around (i.e. king is adjacent to ace). A solution is a sequence of 51 valid moves.

Our model of blackhole solitaire contains reifyimplied less-than constraints ($r \Rightarrow x_1 < x_2$). The less-than constraint places two static triggers, one on the lower bound of x_1 and the other on the upper bound of x_2. SSG always returns two watched literals, the lower bound of x_1 and the upper bound of x_2. When bounds are restored on backtracking, the watched literals are no longer on the bounds. This effect allows SSG to be called many fewer times than CU on these benchmarks. The model also contains reified less-than and sum-greater constraints, which were propagated statically.

As shown in Figure 3 the total number of calls to SSG for all constraints is much smaller than the number of calls to CU for each instance of blackhole we tried. The black line on the plot is the line $y = x/10$, or the "10 times better line", since all points beneath the line use at least 10 times more calls to CU than SSG, for static and watched reifyimply respectively. Using a profiler, we have discovered that the mean number of CPU instructions in a call to SSG was 54 versus 9 instructions per call to CU, meaning that the ratio of CU to SSG would have to be more than 6 for dynamic reify-imply to have a chance of winning. Figure 4 shows that even a ratio of 10 is not sufficient, as the static algorithm is slightly faster on this benchmark.

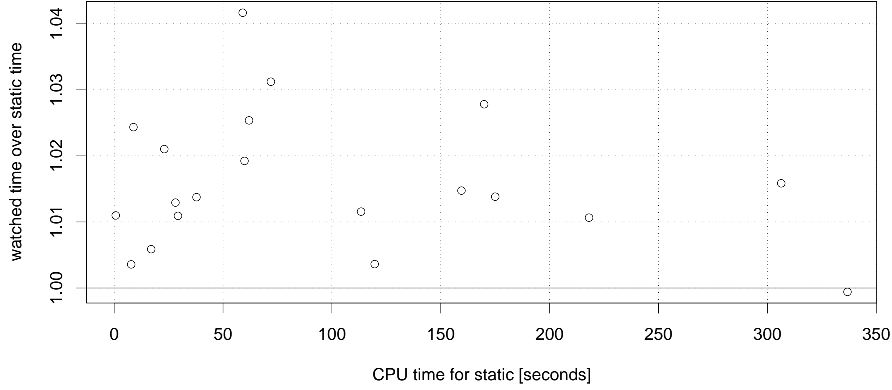


Figure 4: Comparison of time spent on blackhole problems

5.7.3 Contrived benchmark

We use a reified allDifferent constraint in a contrived problem intended to demonstrate the potential of watched reification. We expect that watched reification will perform well if the watched literals can settle on values which are never (or only rarely) removed. This effect was observed for watched reifyimply, on the steel mill slab design problem.

Problem instances can be generated for any positive integer k, and consist of two k-vectors X and Y with domains $\{1, \ldots, k\}$. The constraints are as follows.

- $\forall i \in 1, \ldots, k \; : \; (2 \times X[i]) \neq Y[i]$,
- $X[k-1] \neq X[k]$,
- $X[k-1] = X[k]$, and
- $r \Leftrightarrow$ allDifferent(Y).

The allDifferent constraint uses a GAC algorithm [23], and maintains a matching from variables to distinct values. SSG for the positive constraint returns a k-matching if one exists, hence there is one watched literal for each variable in X. For the static reify, CU is called for any domain change. CU is very similar to SSG, it maintains a maximal matching using the same algorithm as SSG.

The negative constraint waits until all variables are assigned, then checks the assignment[3]. SSG for the negative constraint places two watched literals on different

[3] The standard implementation in Minion 0.8 is a Watched OR of equal constraints on all pairs of variables. Unfortunately, Watched OR is incompatible with the static reification algorithm, so for this experiment the Watched OR was replaced with an assignment checker.

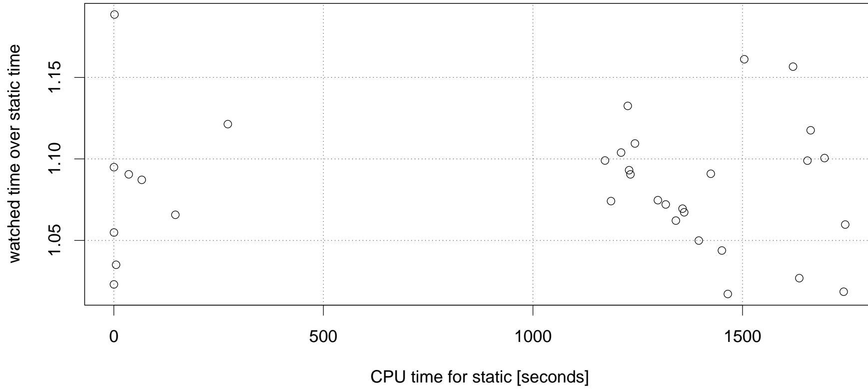


Figure 5: Comparison of time spent on English peg solitare instances

values of an unassigned variable, if possible. If all variables are assigned, SSG checks if the constraint is disentailed. CU requires an assignment trigger on each variable.

The variable ordering is X in index order, values are branched in ascending order. $X[k]$ cannot be consistently assigned, and there is no restriction on the rest of X, so the solver explores k^{k-1} assignments of $X[1 \ldots k-1]$. Whenever a variable $X[i]$ is set to j, $2j$ is removed from $Y[i]$ by the not-equal constraint. Therefore odd values in Y are never removed, and watched literals may settle on them.

We ran instance $k = 20$ with a node limit of 10,000,000. Watched reify made 2509 calls to SSG, compared to 10526315 calls to CU. With static reify, Minion took 50.82s, and with watched reify it took 50.16s. Using the *callgrind* profiler (and a node limit of 500,000), we found that Minion uses 6.60 bn CPU instructions with static reify and 6.42 bn with watched reify. The static reify propagator alone uses 193m instructions, compared to 7.90m for the watched reify propagator. This clearly shows that most of the cost is outside the reification, and that watched reify is performing much better than the static variant, as we would expect from the call counts.

5.7.4 English Peg Solitaire

Finally we consider the game of English peg solitaire [24], which is played with 32 pegs placed in a board with 33 holes. Pegs are removed by hopping moves (similar to checkers/draughts) until a goal state is reached or no moves are possible. We use model C of Jefferson et al [24], slightly adapted to suit Minion rather than ILOG Solver. These benchmarks contain a large number of reified sum constraints. The constraints state that a sum of Boolean variables is 1 or more. The length of the sum ranges from 1 to 8 variables.

We used eight instances with different goals. All instances are run to completion. Figure 6 shows that the number of calls to SSG by watched reification is almost exactly

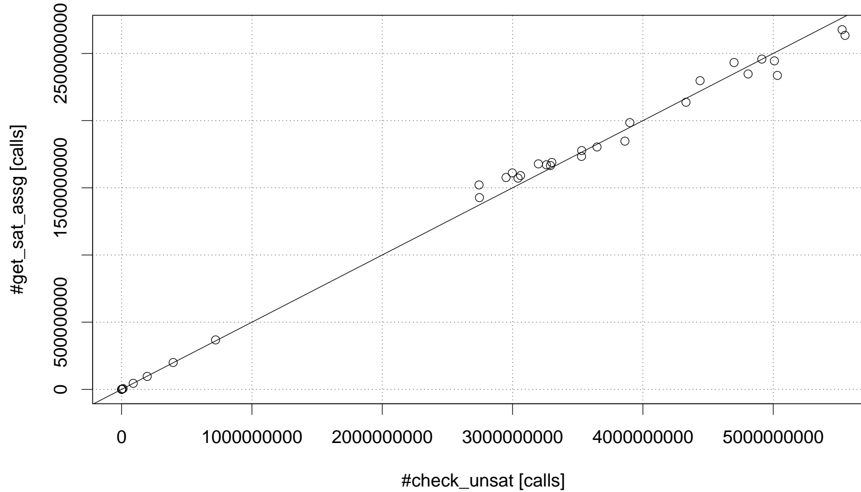


Figure 6: Comparison of calls to SSG and CU for English peg solitaire instances

half the calls to CU for all instances. However, Figure 5 shows that static reification is faster for all instances.

We used the profiler *callgrind* with instance `solitaire_benchmark_6` (which takes 40s with static reification). Minion uses 71.2 billion CPU instructions with static reification, and 78.1 with watched. Static reify alone uses 21.8 bn, and watched reify uses 26.5, an increase of 22%[4].

5.7.5 Conclusion to empirical comparison

The results of our experiments are not conclusive, demonstrating that different implementations perform better on different constraints and problems. In all cases, we have shown the potential of a watched literals approach, by demonstrating that the SSG function is called much less often than CU. On the other hand, static reification (and reifyimply) is simple and fast, and in many cases it is faster than the watched variant.

6 Conclusion

In this paper we have explored possibilities for implementing logical connectives in a constraint solver, with the overall hypothesis that movable triggers and constraint trees together are invaluable. These two solver features are combined with satisfying set generators, which provide an efficient way of checking the satisfiability of a constraint.

[4]Changing the reification algorithm changes the propagation order and affects other constraints. In this case, the difference for reify alone is 4.8 bn and for the whole solver it is 6.9 bn.

First we focussed on OR, ATLEASTK and AND of arbitrary constraints. The ubiquitous way of modelling these in CP is by reifying the constraints, and applying a sum-$\geq k$ constraint (or equivalent) to the reification variables. With this approach, the solver is required to propagate all reified constraints at all times. By contrast, the Watched OR algorithm we present has at most two active constraints at any time — all others have zero cost. Using this approach, we were able to demonstrate a 10,000 times speedup on some instances, and to be consistently much faster than reification.

We also presented Watched ATLEASTK, a generalization of Watched OR, and evaluated it on a Hamming codes problem. In some cases we saw a speedup of over 2,000 times compared to reification.

By implementing satisfying set generators for Watched OR and friends, these parent constraints can be arbitrarily nested, giving a rich language for logical expressions. We hope to extend this work to other logical connectives, and also to achieve GAC in the case where child constraints share variables, while maintaining high performance.

Secondly, we investigated two ways of implementing both reification and reifyimply for any constraint. We described simple algorithms which use static triggers, and more sophisticated algorithms which make use of watched literals to reduce the number of constraint checks. In our experiments, the results were mixed. In some cases, the simple static algorithms were faster, and in others the watched algorithms paid their additional overhead and were more efficient.

The common thread through this paper is that watched literals, satisfying sets and constraint trees together allow simple, efficient implementation of logical connectives of constraints. Once a constraint has a satisfying set generator (which is usually much simpler than its propagation function), it can be used in Watched OR and other parent constraints, and it can be reified and reifyimplied. This makes a simple, general and compelling framework for implementing logical connectives.

Acknowledgements

We would like to thank anonymous reviewers for their helpful comments about an earlier version. This work was funded by EPSRC research grant EP/C523229/1 (Jefferson), EPSRC research grant EP/E030394/1 (Moore, Nightingale), and a Royal Society Dorothy Hodgkin Fellowship (Petrie).

References

[1] Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: Conf. ECAI 2006, IOS Press (2006) 98–102

[2] Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in Minion. In Benhamou, F., ed.: In conf., CP 2006. Volume 4204 of LNCS., Springer (2006) 182–197

[3] Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)

[4] Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier (2006)

[5] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: conf. DAC 2001, ACM (2001) 530–535

[6] Müller, T., Würtz, J.: Constructive disjunction in Oz. In: WLP. (1995) 113–122

[7] Würtz, J., Müller, T.: Constructive disjunction revisited. In Görz, G., Hölldobler, S., eds.: In conf. KI-96. Volume 1137 of LNCS., Springer (1996) 377–386

[8] Lagerkvist, M.Z., Schulte, C.: Propagator groups. In: Proceedings 15th International Conference on Principles and Practice of Constraint Programming (CP 2009). (2009)

[9] Bacchus, F., T.Walsh: Propagating logical combinations of constraints. In Kaelbling, L.P., Saffiotti, A., eds.: In conf. IJCAI-05, Professional Book Center (2005) 35–40

[10] Lhomme, O.: An efficient filtering algorithm for disjunction of constraints. In: Conf. CP 2003, Springer (2003) 904–908

[11] Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In: Proceedings CPAIOR 2004. (2004) 209–224

[12] Hentenryck, P.V., Saraswat, V., Deville, Y.: Constraint processing in cc(fd). Technical report, Brown University (1991)

[13] Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings 12th National Conference on Artificial Intelligence (AAAI 94). (1994) 362–367

[14] Aggoun, A., Chan, D., Dufresne, P., Falvey, E., Grant, H., Harvey, W., Herold, A., Macartney, G., Meier, M., Miller, D., Mudambi, S., Novello, S., Perez, B., van Rossum, E., Schimpf, J., Shen, K., Tsahageas, P.A., de Villeneuve, D.H.: Eclipse user manual release 5.10 (2006) http://eclipse-clp.org/.

[15] Schulte, C.: Programming deep concurrent constraint combinators. In: Proceedings of Practical Aspects of Declarative Languages (PADL 2000). (2000) 215–229

[16] Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of global constraints. Constraints **12(2)** (2007) 239–259

[17] Jefferson, C.: Representations in Constraint Programming. PhD thesis, University of York (2007)

[18] Daniel, P., Semple, C.: Supertree algorithms for nested taxa. In Bininda-Emonds, O., ed.: Phylogenetic Supertrees: Combining information to reveal the tree of life. Computational Biology Series Kluwer (2004) 151–171

[19] Gent, I., Prosser, P., Smith, B., Wei, W.: Supertree construction with constraint programming. In: Principles and Practice of Constraint Programming, Springer (2003) 837–841

[20] Moore, N.C., Prosser, P.: The ultrametric constraint and its application to phylogenetics. Journal of Artificial Intelligence Research **32** (Aug 2008) 901–938

[21] Frisch, A.M., Miguel, I., Walsh, T.: Modelling a steel mill slab design problem. In: IJCAI-01 Workshop on Modelling and Solving Problems with Constraints. (2001) 39–45

[22] Gent, I.P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B.M.: Search in the patience game 'black hole'. AI Communications **20**(3) (2007) 211–226

[23] Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. Artificial Intelligence **172**(18) (2008) 1973–2000

[24] Jefferson, C., Miguel, A., Miguel, I., Tarim, A.: Modelling and solving english peg solitaire. Computers and Operations Research **33**(10) (2006) 2935–2959