Efficient Propagation of Disjunctive Constraints using Watched Literals

Christopher Jefferson and Karen E. Petrie

Computing Laboratory, University of Oxford, UK, email:
chris.jefferson@comlab.ox.ac.uk, karen.petrie@comlab.ox.ac.uk

Abstract. Efficient constraint propagation is important for all constraint solvers. One of the key reasons that Boolean Satisfiability solvers are often more efficient than constraint solvers is that they can propagate disjunctive constraints efficiently. This propagation is undertaken using the watched literals mechanism. In a recent paper Gent et al. showed how watched literals can be used in constraint programming, to propagate simple disjunctive constraints. This paper shows through experiment the deficiencies in the existing methods of implementing the disjunction of a set constraints in CP solvers and presents a new algorithm based on watched literals which when applicable massively outperforms all existing algorithms. In particular we show that if variables are not duplicated between the different constraints being considered disjunctively, and a Generalised Arc Consistent (GAC) propagator is known for each of the constraints; then GAC propagation can be efficiently achieved across the entire disjunctive constraint. We end by showing how this new propagation mechanism can be used to produce efficient models of combinatorial problems, including that of finding Hamming codes.

1 Introduction

A *constraint satisfaction problem* (CSP [1]) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. 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. Constructing a problem into this CSP framework is the art of *modelling*. A *solution* to a CSP is an assignment to all the variables that satisfies all the constraints. Solutions are found for CSP's through systematic search of possible assignments to variables. During search constraint *propagation* algorithms are used. These propagators make inferences, recorded as domain reductions, based on the domains of the variables constrained and the assignments that satisfy the constraints. If at any point these inferences result in any variable having an empty domain then search backtracks and a new branch is considered.

This paper outlines a new propagation algorithm, for the disjunction of a collection of constraints i.e. the case where you need to ensure that at least one constraint from a collection is imposed. This propagation method makes it easy to find an efficient model for problems which include a disjunction between

constraints. This work is motivated by Example 1, which was proposed by a researcher in a field other then that of CSP.

Example 1. How do you model efficiently as a CSP that two arrays, X and Y, of equal length n, are not the same i.e. $\exists i.\ X[i] \neq Y[i]$

There are two ways to model this using the constraints built into Minion [2]. The first method introduces auxiliary variables as an array of Booleans, B of length n. It makes use of two constraints; firstly a reified version of the not equals constraint (*neq*) and secondly the sum greater then or equal to constraint (*sumgeq*). The reified *neq* constraint takes in two variables and returns a Boolean specifying whether these two variables are not equal. The *sumgeq* constraint specifies that the sum of an array must be more then the given value. The CSP model is then given by the constraints:

$\forall i.\ \mathbf{reify}(\mathbf{neq}(X[i], Y[i]), B[i]), \mathbf{sumgeq}(B, 1)$

This method provides generalised arc consistent (GAC) propagation, which is the strongest form of propagation. GAC propagation guarantees that the domains of all the constrained variables are reduced as far as possible at every node of the search tree. However, this is done at a cost as the number of search nodes processed per second for this method is poor. Full experimental results for this method can be seen in Section 5.1.

The problem can also be modeled a second way using the *element* constraint, as well as the *neq* constraint which the previous method used. An efficient propagator for the *element* constraint was devised by Gent et al. [3], using watched literals for propagation. Watched literals are the method used by Boolean Satisfiability (SAT) solvers to provide efficient propagation. This propagation method is explained in more detail in Section 2.1. The element constraint takes an array of variables X and a single variable i, it returns the value x of the array variable at index i i.e. $X[i] = x$. The CSP model is then:

$\mathbf{element}(X, i, x), \mathbf{element}(Y, i, y), \mathbf{neq}(x, y)$

This method runs faster then the previous model in that it processes more search nodes per second. However, it does not obtain 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 4.2.

One obvious way to express this problem is as $(X[1] \neq Y[1]) \vee (X[2] \neq Y[2]) \vee \ldots \vee (X[n] \neq Y[n])$, where the arrays are both of length n. This disjunct of constraints looks a lot like a SAT problem, so it is an obvious step to treat it as such and use the SAT propagation method of watched literals to provide efficient and fast propagation. Being able to propagate all constraints which take the form of a disjunct of smaller constraints is the focus of this paper. Our watched literals based method runs at a higher speed then either model explained above and provides GAC propagation for this problem, making it a more efficient method then either of the models above.

In the next Section of this paper, we go on to explain in more detail the background of this method, including a detailed explanation of how watched literals work. The subsequent section provides a detailed account of our method, including outlining the algorithm in full and proving which cases obtain GAC (the

highest level) of propagation. We then finish by looking at some experimental results on a number of problems, including finding Hamming Codes.

2 Background

The success of constraint programming is based upon its flexibility and expressivity, constraints provide a rich language allowing most problems to be expressed straightforwardly once the model is selected. However, constraint programming has a flaw, the expressibility of the language means that in order for a constraint program (CP) to run efficiently an expert in the field has to choose the correct set of constraints to use; what level of propagation should be applied on each constraint; which order the variables should be searched in; and which search procedure to undertake. The result is that constraint programming, for any problem which is difficult to solve, requires a high degree of expertise and is outside the scope of all who have not received a great deal of training in the field. Puget [4] suggests that today's constraint toolkits are far too complex to achieve widespread acceptance and use. He advocates a 'model and run' paradigm for CP similar to that achieved by SAT and mathematical programming.

Minion [2] is a fast and scalable CP based on the model and run paradigm. A number of the early design decisions are modelled on those of zChaff, which revolutionised modern SAT solving [5]. In a later paper Gent et al. borrowed another feature from SAT, in the form of watched literals. Gent et al. showed how watched literals can be used to implement simple disjunctive constraints along with more complex constraints, such as the element constraint mentioned in the previous section. This paper builds on that work by showing how watched literals can be used to efficiently propagate more complex constraints, namely where each of the disjuncts is itself a constraint. This adds to the ethos of Minion in two ways. Firstly, such constraints often seem to occur in practice, so giving end users a guaranteed efficient implementation means that decompositions (of the type shown in the Introduction) do not need to be considered. It is now possible to choose any Minion constraint and disjunct this with any other constraint, this will be propagated in a very efficient manner. This adds to the ease of use of the solver and falls within the model and run paradigm. Secondly, we are continuing in the Minion ethos of seeing which elements can be taken from SAT to increase the efficiency of CP solvers.

It should be noted however, that although the method outlined in this paper was conceived as part of the Minion ethos, it is actually a general method that can easily be implemented in any constraint solver where watched literals are present. As watched literals are fundamental to the performance of our algorithm, we shall now describe them in depth, using SAT constraints as an example.

2.1 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's 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's 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 [3].

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

Backtrackable: 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 2 demonstrates how watched literals are traditionally used to implement SAT.

Example 2. 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 literals could be assigned true, no propagation can occur. Once only one literal is satisfiable, it must be true. Therefore at all points the algorithm watches two literals, and when only one still holds it is assigned true. In the table the literal assigned through search at that level is highlighted and the last column explains the effect of this change on the watched literals.

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}	D	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}	{0,1}	D	B	No effect - triggers left on D and B.

There are a number of important points to notice about Example 2. 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.

2.2 Related work on disjunction in constraint programming

Given the importance of having efficient disjunctive constraints, as we have highlighted in the previous sections, it is unsurprising to note that there has been previous research papers in this area. There are three research projects which have tackled the same problem as ourselves, that of a disjunct of constraints.

The first by Müller and Würtz [6, 7] implements disjunction between constraints in Oz without the use of watched literals. This method always gets GAC propagation, including the case where there are repeated variables between disjuncts, which our method does not. This is explained in Section 4. However, it has similar performance to the sum decomposition model, which is the first model outlined in Example 1, so our method always manages to search more nodes per second. We will show in Section 5 that in practice our method is often at least 10 times faster then this and never slower. Also it should be noted that in any solver where watched literals are implemented, our method is very easy to execute. Whereas the method of Müller and Würtz requires a detailed infrastructure of its own.

The second paper in this area by Bacchus and Walsh [8] provides theoretical results for considering disjunction between constraints, but no implementation framework details are given. Hence, it is difficult for us to see how our work relates in practice. However, we do note that no mention is given of watched literals in this paper so our algorithm is very different in nature. In the next section we go on to describe exactly how our algorithm operates.

Lhomme [9] completed the third work in this area. Their paper shows how finding supports in algorithms like GAC-Schema can be improved by making use of the fact constraints are a disjunction. This massively improves performance over treating the whole constraint as a set of tuples. However, the method described in Lhomme's paper still keeps an assignment to all the variables in the disjunction for each literal in each variable.

3 Overview of Method

The general design of our algorithm for propagating complex disjunctive constraints can be seen as an extension and generalisation of the basic algorithm for implementing SAT. Both the general SAT algorithm, and our algorithm, follow the same basic outline:

Setup Phase: Find two disjuncts which have a satisfying assignment. If two can be found, then watch a satisfying assignment to both disjuncts, else move to the **Propagation phase**.

Update Phase: If a domain value being watched is removed then look for another satisfying assignment to the same disjunct, or a satisfying assignment to a different disjunct. If one can be found watch that, else go to the **Propagation Phase**

Propagation Phase: If only one satisfiable disjunct exists, propagate that it is true, else the constraint fails.

This basic algorithm makes use of the fact (proved in Theorem 1 shown below) that during search, given a constraint of the form $C_1 \vee C_2 \vee \ldots \vee C_n$, where no two disjuncts share variables, then there is no need for propagation as long as two disjuncts have a satisfying assignment. There are however two features of SAT which make implementing this framework simpler, then it is for our more general algorithm.

The first simplification is that given a disjunct of a SAT constraint, checking if a satisfying assignment exists is trivial, whereas for a general disjunct this can be more complicated. The second issue comes in the **Propagation Phase**. The SAT constraint is propagated by assigning the variable either TRUE or FALSE and after this every variable in the constraint is assigned. Consider however a constraint with the disjunct $x \neq y$, where the current domain of both x and y is $\{0, 1\}$. Given these domains no propagation can occur, as every domain value for both variables is part of a solution to the disjunct. However, once x is assigned 0, then y must be assigned 1.

This means that propagation may have to be performed more then once on a disjunct once we enter the **propagation phase**. This means in our algorithm the propagation phase is replaced from the one in SAT with:

Propagation Phase: If the 'update phase' ever fails, it means only one of the disjuncts can possibly be satisfied. At this point, start propagating that constraint as normal.

4 Implementation Details

Our algorithm is split into three clear phases as in the overview in the previous section, namely a **setup** phase, a **updating** phase and a **propagation** phase. In this section we will present each phase separately.

The first thing to note is the basic principle behind our algorithm, which is to consider each disjunct in isolation, never removes valid solutions. Theorem 1 provides the central result which ensures the correctness of our algorithm. This theorem shows that assuming that no two disjuncts in our constraint share a variable, then we can implement GAC propagation by waiting until only one disjunct can be satisfied, and then GAC propagating that disjunct.

Theorem 1. *Consider a constraint C which can be expressed as $Con_1 \vee Con_2 \vee \ldots \vee Con_n$ where the scopes of the Con_i are disjoint, and 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 two of the Con_i have satisfying assignments in the D_v*
2. *Exactly one of the Con_i has a satisfying assignment in the D_v, and the D_v are GAC with respect to that Con_i.*

Proof.
1. Assume that Con_i and Con_j have satisfying assignments. Then given any assignment to one variable which is not in the scope of Con_i, a satisfying assignment to C can be generated by assigning the variables in the scope of Con_i so that they satisfy Con_i. For variables in Con_i, the variables in the scope of Con_j can be similarly assigned so that they satisfy.
2. Assume that Con_i is the only satisfiable disjunct for the sub-domains D_v. There is support for all variables in all other disjuncts, as they are in an assignment which assigns the variables in Con_i such that they satisfy Con_i. Any assignment to the variables in the scope of Con_i cannot be true because one of the other disjuncts is true, and therefore all assignments can be extendable to a solution if and only if Con_i is GAC with respect to D_v.

We note that our algorithm, like any propagator, is still correct when a variable occurs in multiple disjuncts by treating the multiple occurrences as if they were distinct variables with the same domain. However, unlike the alternative algorithms presented in [6, 7], it will not achieve GAC propagation.

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. Is reverted when search backtracks.

Disjuncts: The disjunct constraints, named C_1 to C_n. A track is also kept of which disjuncts are currently being watched.

One important feature of our algorithm is that while it requires a propagator for each disjunct, this propagator can be any standard propagator, either using watched literals or other standard methods. This ensures that propagating a disjunct is almost as efficient as propagating a normal constraint.

Unlike the algorithm presented in [6, 7], there is no requirement for any changes to be made to how the domains of variables are queried or changed. However, it is necessary to add a simple check to the part of the constraint solver responsible for activating constraints. This is the **propagating** Phase of the algorithm.

if *PropagateMode* = FALSE **then**
    do nothing;
else
    propagate constraint as normal;
end

This code is required because in the middle of search when only a single disjunct constraint can be satisfied standard propagation is applied to this constraint. However, this propagator must be removed when search backtracks to the node where the propagator was added. In some frameworks this removal of triggers on backtrack occurs automatically, but the major strength of watched literals is that they require no work to be performed on backtrack. Therefore this additional check must be added.

As well as the propagator for each disjunct constraint, another function is required which will return a valid assignment to the disjunct, or return that no valid assignment can be found. For a large number of constraints this is a much simpler and shorter piece of code then a general propagator. Assuming these functions are in place the method then commences by initiating the **setup** phase. This searches for two satisfiable disjuncts. If two can be found then they are both watched, if one is found then PropagateMode is entered, if none are found then the constraint fails.

PropagateMode = FALSE;
if $\exists i.\ C_i$ *is satisfiable* **then**
  if $\exists j.\ i \neq j \wedge C_j$ *is satisfiable* **then**
    Add watches to satisfying assignment of C_i;
    Add watches to satisfying assignment of C_j;
  else
    addPropagationTo(C_i);
    PropagateMode = TRUE;
  end
else
  Fail;
end

Now we enter the central loop of the method, the **updating** phase which is called whenever the satisfying assignment to a disjunct is lost. The most important feature of this phase to notice is that in the case when no unwatched satisfiable clause can be found, the watches are not removed from any disjunct, including the one which just lost support. This is because while **PropagateMode** is true, any calls to these triggers are ignored. When search eventually backtracks to the start of the node in which **PropagateMode** was set to true, then all these watches will become valid again.

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

Lemma 1. *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 invarients are true.*

1. *Either* **PropagateMode** = TRUE *or two satisfying assignments in two disjuncts are being watched.*
2. *Either* **PropagateMode** = FALSE *or only one clause is satisfiable and that clause is being propagated.*

```
Input: C_1, C_2, ..., C_n: disjuncts
Input: d : Disjunct which failed
Global Data: PropagateMode
if PropagateMode then
    Return
end
if C_d is satisfiable then
    Move watches to new satisfying assignment to C_d;
else
    if ∃i. C_i is satisfiable and unwatched then
        Move watches to satisfying assignment to C_i from C_d;
    else
        Remove all old Propagation trigger / setup on all disjuncts;
        Initialise Propagation on the other watched disjuncts;
        PropagateMode = TRUE;
    end
end
```

Proof. 1. Clearly this 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 satisfied. There are two other cases to consider:
   - Backtrack from a node where **PropagateMode** is FALSE. In this case **PropagateMode** must still be FALSE. While the watched assignments might not be same one as when this search node was left, if they were allowed by children of this node they are supported here.
   - Backtrack from a node where **PropagateMode** is TRUE to one where it is FALSE. In this case the watched assignments have no changed since this node was left, and as they were allowed before they are still allowed now.
2. When **PropagateMode** becomes TRUE, propagation progresses as normal. Further, when **PropagateMode** was made TRUE only one clause was satisfiable and therefore it is not possible that more then one clause could be satisfiable deeper in search. When search backtracks to the node at which propagation was set up, **PropagateMode** becomes FALSE and all the propagation triggers are disabled and will be removed before **PropagateMode** is ever made TRUE again.

Lemma 1 proves that our algorithm works correctly, when moving between the **Updating** phase and the **Propagation** phase. In this next section we go on to show a generalisation of our algorithm that allows more complex constraints to be constructed.

4.1 Implementing Watched Sum

The constraint $C_1 \vee \ldots \vee C_n$ can also be expressed as the constraint $\sum_i C_i \geq 1$ if Boolean expressions are treated as taking the value 0 if false and 1 is true. In

a similar fashion to how the "watched sumgeq" presented in [3] generalised the basic SAT algorithm, we can also easily generalise our algorithm to implement $\sum_i C_i \geq j$ for a constant j. In practice this algorithm is most efficient when j is small compared to the number of disjuncts.

The general design of this more generalised algorithm is similar to the basic algorithm presented at the start of this Section. Rather then watching two disjuncts and propagating the single disjunct if the other loses support, we instead watch several disjuncts, and propagate all of those remaining once one of them loses support. The proofs of correctness follow almost identically, as does the algorithm itself.

The algorithm for the **setup** phase from the previous section, is extended to the algorithm below:

Input: $C_1, C_2, \ldots, C_n$: disjuncts
Input: j: number of disjuncts to watch
Global Data: PropagateMode
PropagateMode = FALSE;
propagatingDisjunct = -1;
if $\exists z_1, \ldots, z_j.\ \forall i \neq j.\ z_i \neq z_j \wedge C_{z_i}$ *is satisfiable* **then**
    if $\exists j.\ i \neq j \wedge C_j$ *is satisfiable* **then**
        Add watches to satisfying assignment to all the C_{z_i};
        Add watches to satisfying assignment to C_j;
    else
        add Propagation to all the C_{z_i});
        PropagateMode = TRUE;
    end
else
    Fail;
end

4.2 Theoretical Analysis of Alternative Implementations

Our algorithm in not the only method of implementing disjunction. In this section we will discuss two pre-existing methods outlined in Example 1 which use standard disjunction and element constraints.

The first, model from Example 1 implements disjunction by flattening the constraint by the introduction of extra variables. This is the 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.

The second model from Example 1 uses the element constraint which implements $X[i] = x$. There are two distinct problems with implementing this as $X[i] = x, Y[i] = y, x \neq y$. Consider the following example:

$$X[0] = \{0\}\ X[1] = \{1\}$$
$$Y[0] = \{1\}\ \ Y[1] = \{0\}$$

In this case, x and y can still take any value, as can the index variable i. Further, $x \neq z_y$ leads to no propagation. Therefore not only does this representation not achieve GAC but even after assigning all variables in X and Y the extra variables must still be assigned before failure can occur.

The second problem comes from the fact that there may be many indices i where $X[i] \neq Y[i]$. In this case, a different solution will be generated for 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. In theory this problem could be fixed by symmetry breaking, but the simplest way of doing so would involve introducing all $X[i] \neq Y[i]$ for each index value i, removing the advantage of this more compact representation.

5 Experimental Results

To test our new algorithm we performed three experiments. These three problems will show our algorithm and the existing alternatives in various comparisons.

5.1 The Generalised Pigeon-Hole problem

The first experiment we performed is that of $\exists i.\ X[i] \neq Y[i]$ which is outlined in detail in Example 1. This is actually 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.

Watched OR: Implemented as a watched OR, the algorithm described in this paper.
Element: The second model from Example 1.
Sum: The first model from Example 1.
Watched Sum: The same algorithm as **sum**, except the sum constraint is replaced by a watched SAT clause $b[1] \vee \ldots \vee b[l]$.
Custom: A custom-written traditional propagation algorithm. This algorithm is designed to supersede the algorithm of Lhomme in [9].

Note that using Theorem 6.6 from [10], as long as we get GAC on each of the constraints in the **Sum** and **Watched Sum** models, we get GAC over the whole problem, 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**.

In fact, it turns out that this model does indeed produce larger searches. Consider the following small instances of the array pigeon-hole problem.

	Element		Watched OR	
	Time	Nodes	Time	Nodes
8 arrays of length 3 and domain size 2	20.25	12,335,593	0.05	25
8 arrays of length 4 and domain size 2	1716.11	1,092,789,218	0.05	33

These instances show how poorly the **Element** model performs and therefore 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.

We compare the number of nodes per second achieved on average over 100 seconds on various instances of the array pigeon-hole problem, always considering the case of finding 100 arrays.

Length	Domain	Watched-OR Size	Flattening + traditional sum	Flattening + watched sum	Traditional Single Propagator
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

These results show the massive improvements which can arise from using the Watched-OR propagator. The comparison with the traditional single propagator shows the huge gains which can be achieved by watching only a small proportion of the variables at any time. This leads to the perhaps surprising result that the Watched-OR algorithm sometimes increases in speed as instance size increases, as the proportion of the variables being watched decreases.

The reason for the instability in the performance of the single traditional propagator is unclear. However it clearly wins over the decompositions while also clearly being beaten by the Watched-OR algorithm.

The massive slowdowns of the flattening methods arise from the much larger numbers of variables required to implement them. The small differences between using a watched or traditional sum show that the gain from using watched literals is almost completely removed if flattening is used.

5.2 The Anti-Chain problem

The second experiment we consider is the anti-chain problem, defined below.

Definition 1. *An **anti-chain** is a set S of multisets where $\forall\{x, y\} \subseteq S.\ x \not\subseteq y \wedge y \not\subseteq x$. We consider the following class of problems:*

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

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, the algorithm described in this paper.

Element: Introduce 3 variables, a with domain $\{1, \ldots, l\}$ and b and c, each with domain $\{1, \ldots, d\}$. Impose the three constraints $M[a] = b, N[a] = c$ and $b < c$.

Sum: Introduce a new array of Boolean variables $b[l]$ and impose the set of constraints $\forall i \in \{1, \ldots, l\}.\ (M[i] < N[i]) \leftrightarrow b[i]$. Then impose $\sum(b_{ij}) \geq 1$. We believe this algorithm is similar to the one given in [6, 7], although simplified as it only requires disjoint variables.

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

In this, and later problems we did not construct a single propagator due to lack of time and lack of an efficient implementation of Lhomme's algorithm from [9].

Similarly the previous experiment, the **Watched OR**, **Sum** and **Watched Sum** all achieve GAC propagation. 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.

Instance	Element		Watched Or	
	Time	Nodes	Time	Nodes
12 arrays of length 4 and domain size 3	63	3,030,555	7.98	2,189,034
11 arrays of length 5 and domain size 2	29.34	2,411,733	7.91	2,411,733

These results are much more competitive then those in the pigeon hole problem, on some instances the **element** model achieves the same sized search as our algorithm. However, the **element** algorithm always loses out both on number of nodes searched per second, and occasionally the search size increases massively. Furthermore, it still produces much larger numbers of solutions, for example on the 6 arrays of length 4 and domain size 2 instance the **element** model finds 46,080 solutions in 7 seconds, while the **watched OR** model find 720 solutions in 0.2 seconds.

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 of varying

length and domain size. We do not solve the entire problem as the longer than 1 hour for all problems, we also know the number of nodes and search will be identical.

Length	Domain Size	Watched-OR	Flattening + traditional sum	Flattening + watched sum
5	2	47,970	3,285	3,137
10	2	38,316	2,793	2,402
15	2	32,544	2,416	2,031
20	2	28,550	2,028	1,771
25	2	24,835	1,808	1,506
30	2	21,858	1,627	1,288
35	2	20,090	1,449	1,153
40	2	18,250	1,226	979
45	2	17,087	1,052	839
50	2	15,820	888	716
5	10	1,467	77	70
10	10	1,228	71	64
15	10	1,286	70	63
20	10	1,330	70	65
25	10	1,357	70	64
30	10	1,487	69	65
35	10	1,698	70	67
40	10	1,971	72	69
45	10	2,190	75	72
50	10	2,249	76	72

A number of conclusions can be drawn from these results. First of all, our algorithm performs well 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 22 times for length 50. We note that for larger domains the nodes per seconds increases as the problem size increases. This is not a mistake, and appears to arise from the decreasing frequency chance of a conflict occurring. While using a watched sum in the flattening is consistently slightly better, the improvement is nowhere near the gain from our algorithm.

5.3 Hamming Codes

The final experiment considered is Hamming codes, defined below.

Definition 2. *The $\langle n, l, d, s\rangle$ instance Hamming problem is the following CSP:*

Find n arrays of integers, named $M_1, \ldots, M_n$, each of length l and domain $\{1, \ldots, d\}$ which satisfy the constraints:
$\forall i, j \subseteq \{1, \ldots, n\}.\ (\sum_{k \in \{1,\ldots,l\}} M_i[k] \neq M_j[k]) \geq s.$

We test 3 implementations of the constraint: $(\sum_{i \in \{1,\ldots,l\}} M[i] \neq N[i]) \geq s$

Watched OR: Implemented as a watched OR with count, the algorithm described in Section 4.1.

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

Watched Sum: The same algorithm 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, as the performance was so poor it was impossible to usefully compare it to any of the other models.

Here we consider calculating Hamming codes for 50 arrays of Boolean variables, each of length 50. We run this experiment for various different Hamming distances. We do not give the total times and number of nodes as we the total node will be identical and solving time was over the timeout imposed in all cases.

Distance	Watched-OR	Flattening + traditional sum	Flattening + 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

We might expect that flattening and traditional sum would be unaffected by the varying distance. However, there is an obvious decrease in the difficulty of solving problems as the Hamming distance reaches the middle values. Ignoring this effect, we see that for maximum Hamming distances our algorithm actually performs slightly worse that watched sum. This is not surprising, as one of the major benefits of watched algorithms is that they can attach triggers to only a small number of the variables. However it's performance is still competitive as there is no need to introduce auxiliary variables.

6 Conclusion

We have shown the weaknesses of the existing methods of modelling disjunction in constraint programming. To help rectify this problem, we have presented the framework for propagating disjuncts of constraints using the concept of watched literals. One of the major strengths of our algorithm is that it allows existing propagators to be used for the disjuncts without minimal alteration and no loss of efficiency. We have proved that our method obtains GAC propagation in the case where there are no shared variables between disjunct constraints. We have

also shown that this constraint is useful in modelling multiple problems which include disjuncts and related constraints, such as hamming distance. The use of our constraint makes it possible to solve these problems several orders of magnitudes faster then the current best known methods, including specialised propagators which do not make use of watched literals. We hope to extend this work to other logical connectives, and also to achieve GAC in the case where disjuncts share variables, while maintaining high performance.

Acknowledgments

We would like to thank the referees for their comments which helped improve this paper. We also thank Neil Moore and Martin Green for their helpful comments on earlier drafts. The first author is financed by an EPSRC grant and the second holds a Royal Society Fellowship.

References

1. Dechter., R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
2. 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
3. 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
4. J.-F.Puget: Constraint programming next challenge: Simplicity of use. In Wallace, M., ed.: In conf. CP 2004. Volume 3258 of LNCS., Springer (2004) 5–8
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. 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
9. Lhomme, O.: An efficient filtering algorithm for disjunction of constraints. In: Conf. CP 2003, Springer (2003) 904–908
10. Jefferson, C.: Representations in Constraint Programming. PhD thesis, University of York (2007)