A Theoretical Framework for Constraint Propagator Triggering

Abstract

CSP instances are commonly solved by backtracking search combined with constraint propagation. During search, constraint solvers aim to remove any literals (variable-value pair) that can be shown not to be part of any solution. This literal removal, called propagation, is the beating heart of modern constraint solvers.

A significant proportion of the runtime of propagating constraint solvers is spent running propagation algorithms. Therefore any mechanism for reducing how frequently propagators are called leads directly to significant performance improvements. One family of popular techniques is dynamic triggering – these techniques aim to avoid invoking a propagator when it would remove no literals. While this technique has been successful in practice, it has not yet been studied theoretically. This paper provides a theoretical framework for understanding when dynamic triggering will be successful. In particular, we prove when a literal deletion does not require a propagator to be executed. To achieve this, we describe supports: a support for a constraint is a set of literals whose presence in a search state ensures that propagating the constraint will not remove any literals. Therefore running the propagator when a literal outside the support is deleted is a waste of time.

By characterising supports and giving a definition of dynamic and static supports for the CSP, we provide the framework for a proper analysis. We show how the number of triggers required for different constraints varies widely. For some constraints, dynamic triggering allows very small supports, for others the number of required supports is provably large.

Introduction

Propagation algorithms are an integral part of any modern constraint solver. They improve performance by removing parts of the search that contain no solutions. The efficient implementations of propagators is a vital part of the design of any constraint solver; so this has become a very active research area. A huge body of literature discusses both the theory and practice of efficient propagation algorithms on a range of constraint types. However, there is very little work on how often and in which circumstances constraint propagators should be activated, or 'triggered'. This paper aims to fill this gap in our knowledge of this important subject.

The concept of different triggering types was described by Gent *et al.* in their work on 'watched literals' which showed how different methods of triggering can make huge differences to the performance of solvers. This was a generalisation of the earlier, highly successful work on watched literals in SAT (Moskewicz et al. 2001). Looking through the recent literature we find more evidence of the effect of different triggering mechanisms including a 26 times improvement for the **element** constraint (Gent, Jefferson, and Miguel 2006b), a 60 times improvement for conjunctions of constraints (Jefferson et al. 2010), a 20 times improvement for lexicographic ordering constraints (Jefferson 2011) and a 2 times speed improvement in relation to the HaggisGAC propagator (Nightingale et al. 2013). However, dynamic triggering techniques do not always lead to a performance improvement. Gent et al. (Gent, Miguel, and Nightingale 2008) also showed that advanced triggering methods usually slow propagation of the **alldifferent** constraint, except for when variables have very large domains. Nightingale went on to show that dynamic triggering is not useful for the **global cardinality constraint** (Nightingale 2011). This large body of practical work makes it clear that a better understanding of triggering mechanisms is required to understand how different triggering methods effect performance.

As a concrete (simplified) example, let us consider an instance of the Boolean Satisfiability (SAT) problem with n variables and m SAT clauses, where each clause contains k variables. On average each variable is contained in $m \times k/n$ clauses, and therefore an average of $m \times k/n$ propagators would be invoked when the domain of a variable is reduced during search. In modern SAT solvers (Moskewicz et al. 2001) and constraint solvers such as Minion (Gent, Jefferson, and Miguel 2006a), the solver chooses two variables from each SAT clause and a clause is only propagated when the domain of one of those two variables changes. This reduces the number of propagator calls when the domain of a variable is changed to an average of $m \times 2/n$, a factor of $k/2$ improvement. As SAT instances often have clauses which are hundreds of variables long, this leads to substantial performance improvements.

This paper gives a theoretical framework which allows us to prove when it is safe to trigger propagators less frequently. In particular, we discuss triggers based on the idea of *support*: a support for a constraint is a set of literals (variable-

value pairs) whose presence in a search state ensures that propagating the constraint will not remove any literals and so would be a waste of time. By only invoking the constraint when one of these literals is removed, we can greatly improve solver performance.

In choosing a support to act as a trigger for a constraint, there are some decisions to be made. If a literal in the support is removed during search, the constraint will then be propagated. What then should happen to the support, before the search proceeds? **Static** triggers use supports that are not changed when a literal is removed; they are set up before search starts and must then remain correct in any possible search state that could occur. Such supports have the advantage that they are simple to implement and reason about. Some solvers only support static triggers, or have limited support for moving and removing triggers during search.

Dynamic triggers on the other hand can change the support whenever the propagator is invoked. The Gecode (Gecode Team 2006) and the Minion (Gent, Jefferson, and Miguel 2006a) solvers support both dynamic and static triggers.

This paper provides a theoretical framework for proper analysis and comparison of dynamic and static triggers, and shows how verifying the correctness of triggers of a constraint can be harder than propagating the constraint.

There are a range of other successful methods of reducing the time spent on propagation, these include not invoking propagators when the work they will perform will be non-zero, but not worth the time taken (Katriel 2006; Boisberranger et al. 2013); allowing propagators to hook more directly into the changes made to variables, so they can ignore changes they do not care about (Lagerkvist and Schulte 2007), and considering the effect of a variable change on a group of propagators simultaneously (Lagerkvist and Schulte 2009). In this paper we consider a single question – when does the removal of a single literal not require a propagator to be invoked. This work can be used in combination with these other papers, as a pre-processing check – if no propagation can occur, there is no need to perform any other checks.

In the next section we define the CSP problems and propagators. ThenI, we define a support for a constraint in a search state, show how to find minimum dynamic supports, and show the complexity of verifying the correctness of supports. Finally we discuss the trade-offs between dynamic and static supports in search.

Definitions

Although this paper discusses when propagation algorithms should be triggered rather than how they should work, we still need to define propagation algorithms and how they operate. We begin by defining a CSP:

Definition 1. A **CSP instance**, P, is a triple $\langle V, D, C\rangle$, where: V is a finite set of **variables**; D is a function from variables to their **domain**; and C is a set of **constraints**.

A **literal** of P is a pair $\langle v, d\rangle$, where $v \in V$ and $d \in D(v)$. For any subset $X \subseteq V$ an **assignment** to X is a set consisting of precisely one literal for each variable in X.

Each **constraint** c is a pair $\langle \sigma, \rho\rangle$ where $\sigma \subseteq V$ is a set of variables called the **scope** of c and ρ is the set of **allowed assignments** to σ for c.

We will use the notation $\sigma(c)$ to denote the scope of c and $\rho(c)$ to denote the allowed assignments for c.

Given a constraint c, an assignment A to $\sigma(c)$ **satisfies** c if A is an assignment allowed by c. A **solution** to P is any assignment to V that satisfies all the constraints of P.

In this paper, we consider solving CSP instances using a backtracking constraint solver. Such solvers work with search states:

Definition 2. Let $P = \langle V, D, C\rangle$ be a CSP instance. A **search state** is either the empty set, or a set of literals containing at least one literal for each variable in V. Given a CSP variable X, we define $S_X = \{i | \langle X, i\rangle \in S\}$.

For any search state S, we define an **S-assignment** to be an assignment that contains only literals in S. For any search state S and any constraint c we define the S-assignments of c, denoted $S(c)$, to be those assignments in $\rho(c)$ which are contained in S.

For any set of literals T we define the **variables of T**, denoted $\mathrm{V}(T)$, to be the variables for which T contains at least one literal.

Note that the definition of a search state assumes there is at least one literal for every variable – we assume that the solver is responsible for dealing with the complete wipe-out of a domain, rather than each constraint individually.

Search states can be changed in two ways, by propagation and by branching. The aim of propagation is to achieve some level of consistency for each constraint and to maintain it as the search state changes. Definition 3 gives a basic definition of a propagator. Further discussion of the basic definition of a propagator can be found in (Green and Jefferson 2008).

Definition 3. A **propagator** p_c for a constraint c is a function from search states to search states which satisfies the following conditions for any search states S and T:

Non-increasing: $p_c(S) \subseteq S$.

Solution Preserving: If $A \subseteq S$ is an assignment to $\sigma(c)$ that is allowed by c, then no literal in A is removed by p_c, i.e. $A \in S(c) \implies A \subseteq p_c(S)$.

Detects Failure: If S allows a single assignment to $\sigma(c)$ and this assignment does not satisfy c, then $p_c(S) = \{\}$.

Example 1 gives an example of a propagator for a simple constraint. Even for this constraint we can see there are a variety of different propagators available. Definition 4 defines a relationship between propagators in terms of their strength.

Example 1. Consider the constraint c defined as $X < Y$, where X and Y have initial domain $\{1, \ldots, n\}$. Given a search state S, a propagator p_c for c must not remove any literal $\langle Y, i\rangle$ from S where $i > min(D_X)$, as it is part of a solution, and similarly it cannot remove any literal $\langle X, i\rangle$ where $i < max(D_Y)$. It can remove any (or all) of the other literals in S. The only deletion which must be performed is that $p_c(S) = \{\}$ when $|S_X| = |S_Y| = 1$, and those two literals make an assignment which does not satisfy c.

Definition 4. If p_c and p'_c are two propagators for a constraint c, p_c is **stronger** than p'_c if for any search state S, $p_c(S) \subseteq p'_c(S)$.

Different levels of strength have been defined for propagators; in this paper, we shall be concerned mainly with Generalised Arc Consistency, (GAC_c), the strongest possible propagator for a constraint c, and the Assignment propagator (Assign_c), the weakest propagator for a constraint c.

Definition 5. For a constraint c, search state S and variable $v \in \sigma(c)$, we define the propagator $\mathrm{GAC}_c(S)$ as follows: the literal $\langle v, d\rangle \in \mathrm{GAC}_c(S)$ if and only if there exists some assignment $\pi \in S(c)$ with $\langle v, d\rangle \in \pi$. This is the strongest possible propagator for c.

It is common to refer to a search state S as being "Generalised Arc Consistent (GAC)" with respect to a constraint if $\mathrm{GAC}_c(S) = S$. We will use this terminology in this paper.

In contrast to GAC_c, which guarantees the most propagation, the Assignment propagator (given in Definition 6) is the propagator which performs the least possible propagation.

Definition 6. For any search state S and constraint c, the **assignment propagator** Assign_c is defined as:

- If S contains more than one literal for any variable in $\sigma(c)$, $\mathrm{Assign}_c(S) = S$.
- If S contains exactly one literal from each variable in $\sigma(c)$, then consider the assignment A represented by these literals. If A is allowed by c then $\mathrm{Assign}_c(S) = S$ else $\mathrm{Assign}_c(S) = \{\}$.

Some GAC and Assign propagator examples are given in Examples 2 and 3.

Example 2. Consider the constraint $c = X < Y$, given earlier in Example 1. Then $\mathrm{GAC}_c(S)$ is the union of $\{\langle X, i\rangle | i \in S_X, i < max(S_Y)\}$ and $\{\langle Y, j\rangle | j \in S_Y, j > min(S_X)\}\rangle$. This removes all domain values which are not in a solution.

The assign propagator for c is easier to define. $\mathrm{Assign}_c(S) = S$, unless $|S_X| = |S_Y| = 1$. In this case $\mathrm{Assign}_C(S) = S$ if the single assignment contained in S satisfies $X < Y$, else it is the empty search state.

Example 3. The **element** constraint $M_x = y$ has scope $\{x, y\} \cup \{M_d \mid d \in D(x)\}$. An assignment S is allowed precisely when it contains the literals $\{\langle x, d\rangle, \langle M_d, i\rangle, \langle y, i\rangle\}$ for some d and i. That is, if x is assigned the value d, then M_d must be assigned the same value as y.

Given the CSP $\langle V, D, C\rangle$, where $V = \{x, y, m_1, m_2, m_3\}$, $D(x) = \{1, 2, 3\}$, $D(y) = \{3, 4, 5\}$, $D(m_1) = \{1, 2\}$ and $D(m_2) = D(m_3) = \{1, 2, 3, 4\}$. Further suppose that $c \in C$ is the **element** constraint $M_x = y$.

Given the search state $S = \{\langle v, d\rangle \mid v \in V, d \in D(v)\}$, then GAC_c removes the literals $\langle y, 5\rangle$ and $\langle x, 1\rangle$ from S. $\mathrm{Assign}_c(S) = S$, as x (along with the other variables) has more than one domain value remaining in S.

When considering multiple constraints simultaneously, there are higher levels of consistency than GAC, for example path consistency (Mackworth and Freuder 1985). These higher levels of consistency can be considered as a single propagator on the conjunction of a set of constraints.

Supporting Propagators

In many cases once a propagator has been called, the propagator need not be reinvoked until many more literals have been removed. Therefore, there are potential savings in propagation effort if we can avoid invoking or even considering a propagator when it is unnecessary. This is the case when we can prove that running it would not lead to further literal deletions. We shall do this by creating supports for propagators, which represents when a propagator may have to be called again and more importantly when we can prove it does not have to be called.

We shall pay special attention to both the GAC and Assign propagators, as they are the extremal cases, but our results unless explicitly stated otherwise apply to all propagators. We begin by defining a *support* for a propagator.

Definition 7. Let c be a constraint, p_c be a propagator for c and S be a search state. We say that a set of literals $U \subseteq S$ is a **support** for p_c in S if it satisfies:

For all search states S', where $U \subseteq S' \subseteq S, p_c(S') = S'$.

The purpose of supports is to avoid invoking propagators. Given a propagator p_c, a search state S and a support U, there is no need to call p_c until at least one literal from U has been removed from S. Supports can therefore allow constraint propagation to be much more efficient. The smaller the support, the easier it is to avoid propagation. Conversely, in the worst case, if the support contains every literal in S, it is no help.

Note that the definition of support does not require that once a literal in the support is lost, then propagation will inevitably delete literals.

Example 4. let c be the constraint $X < Y$ with initial domains $\{1, \ldots, n\}$, as in Example 1. Let S be a search state in which $\mathrm{GAC}_c(S) \neq S$, or equivalently the domains are not GAC. It is easy to see that there is no support for the constraint in S: observe that $\mathrm{GAC}_c(S) \neq S$. Now consider a search state S where $\mathrm{GAC}_c(S) = S$.

Now we will consider cases where $\mathrm{GAC}_c(S) = S$. If $max(S_X) < min(S_Y)$, then all assignments in S satisfy c, and so we can use an empty support (or any set of literals). Consider then the case where $max(S_X) \geq min(S_Y)$. Our support must contain a literal $l_1 = \langle Y, d\rangle$ where $l \in S$ and $d > max(S_X)$, to support the literal $\langle X, max(S_X)\rangle$, and similarly a literal $l_2 = \langle X, d\rangle$ where $l \in S$ and $d < min(S_Y)$. The support can also contain any other literals.

Example 5. Let c be the **element** constraint $m_x = y$ and consider the propagator GAC_c for c, as in Example 3. We will demonstrate one class of supports for **element**, from (Gent, Jefferson, and Miguel 2006b).

Let S be a search state in which $\mathrm{GAC}_c(S) \neq S$, or equivalently the domains are not GAC. It is easy to see that there is no support for the constraint in S: observe that $\mathrm{GAC}_c(S) \neq S$. Now consider a search state S where $\mathrm{GAC}_c(S) = S$, and there are at least two literals of x $\{\langle x, i\rangle, \langle x, j\rangle\} \subseteq S$. Consider any set U where:

- $\{\langle v, d\rangle \mid \langle v, d\rangle \in S, v \in \{x, y\}\} \subseteq U$.
- For all $\langle y, d\rangle \in S$, there exists $i \in D(x)$ with $\langle m_i, d\rangle \in U$.
- For all $\langle x, i\rangle \in S$, there exists $d \in D(y)$ with $\langle m_i, d\rangle \in U$.

We want to show that for any search state S' where $U \subseteq S' \subseteq S$, that $\mathrm{GAC}_c(S') = S'$, and therefore U is a support.

For any literal $\langle y, d\rangle \in S$, since U contains a pair of literals $\{\langle M_i, d\rangle, \langle x, i\rangle\} \subset S$, any assignment containing these and $\langle y, d\rangle$ satisfies $m(x) = y$. Similarly U supports any $\langle x, i\rangle \in S$ as U contains some pair $\langle y, d\rangle, \langle m_i, d\rangle$. Finally, all literals of the form $\langle M_i, d\rangle$ will be supported, because there will exist some triple $\{\langle m_j, d'\rangle, \langle x, j\rangle, \langle y, d'\rangle\}$, where $i \neq j$, as there are at least two literals of x in S.

One obvious question is how propagator strength effects the size of supports. Unfortunately, stronger propagators (which in general lead to smaller searches) require larger supports, as Theorem 1 shows. This leads to an inevitable trade-off between propagation strength and support size.

Theorem 1. Given two propagators p_c and q_c for a constraint c, where p_c is stronger than q_c, then if U is a support for the search state S for p_c, then it will be a support for q_c.

Proof. From the definition of support, for all search states S', where $U \subseteq S' \subseteq S, p_c(S') = S'$. As q_c is a weaker propagator than p_c, then $p_c(S') \subseteq q_c(S')$ and $q_c(S') \subseteq S'$ from the definition of propagator, so $q_c(S') = S'$. □

Jefferson (Jefferson 2011) showed that while GAC propagator of the lexicographic ordering constraint on two arrays of length n can require supports of size $2 \times n$, a slightly weaker propagator could achieve supports of size 2. This greatly increased the speed of the solver, as the slightly larger search was outweighed by the decreased propagator invocations.

The following theorem gives a simple characterisation of supports for GAC propagators, which will allow determination of minimum supports.

Theorem 2. Given a constraint c, a set of literals U is a support in S for GAC_c if and only: For any literal $\langle v, d\rangle \in S$ and any S-assignment A to the variables in $\sigma(c) - \mathrm{V}(U) + \{v\}$ which contains $\langle v, d\rangle$, there exists an assignment A' to $\sigma(c)$ where $A \subseteq A'$, $A' - A \subseteq U$ and A' satisfies c.

Proof. Firstly suppose that U is a support in S for c. If U is empty then any search state $S' \subseteq S$ has $GAC_c(S') = S'$ and so every S-assignment is in c. Otherwise, choose any $u \in \mathrm{V}(U)$ and consider a S-assignment A to $(\sigma(c) - \mathrm{V}(U)) \cup \{u\}$. Let $S' = U \cup \{\langle v, d\rangle \mid \langle v, d\rangle \in A, v \neq u\} \cup \{\langle v, d\rangle \mid \langle v, d\rangle \in S, v \notin \sigma(c)\}$. We know that $GAC_c(S') = S'$ as $U \subseteq S'$. We now appeal to the definition of GAC: For any $\langle v, d\rangle \in S'$ where $v \in \sigma(c)$ there exists some assignment $\pi \in S'(c)$ with $\langle v, d\rangle \in \pi$. This is precisely what is required.

We will now prove when the condition of our theorem holds, U is a support. If every S-assignment satisfies c then certainly $GAC_c(S') = S'$ for any search state $S' \subseteq S$ and any U is a support. These are the only constraints where the empty set is a support. Assume therefore $\mathrm{V}(U)$ is not empty and that for any $u \in \mathrm{V}(U)$ and any S-assignment A to $(\sigma(c) - \mathrm{V}(U)) \cup \{u\}$ there exists a U-assignment B to $\mathrm{V}(U) - \{u\}$ such that the assignment $A \cup B$ satisfies c.

Choose any search state S' where $U \subseteq S' \subseteq S$. We have to show that $GAC_c(S') = S'$. Again we appeal to the definition of GAC: For any $\langle v, d\rangle \in S'$ where $v \in \sigma(c)$ we need to show that there exists some assignment $\pi \in S'(c)$ with $\langle v, d\rangle \in \pi$.

There are two cases. Either $v \in \mathrm{V}(U) \cap \sigma(c)$ or $v \in \sigma(c) - \mathrm{V}(U)$. In the first case we can choose any S'-assignment to $(\sigma(c) - \mathrm{V}(U)) \cup \{v\}$ containing $\langle v, d\rangle$ and there exists an appropriate U-assignment to $\mathrm{V}(U) - \{v\}$.

In the second case, since $\mathrm{V}(U) \cap \sigma(c)$ is not empty, we can choose any $u \in \mathrm{V}(U) \cap \sigma(c)$ and any S'-assignment to $(\sigma(c) - \mathrm{V}(U)) \cup \{u\}$ containing $\langle v, d\rangle$ and there exists an appropriate U-assignment to $\mathrm{V}(U) - \{v\}$. □

Examples 6, 7 and 8 uses Theorem 2 to give some minimal GAC supports, showing how some constraints need very small sets of supports while other like **alldifferent** require very large sets.

Example 6. Consider the constraint c defined as $B_1 \vee \cdots \vee B_n$ over n boolean variables $B_1, B_2, ..., B_n$, a search state S and support U.

Is there exists a variable B_i where $\langle B_i, \textsc{False}\rangle \notin S$, then $\langle B_i, \textsc{True}\rangle$ must be in S. This means all S-assignments satisfy c, the constraint is entailed and the support can be empty. Further, if there is only a single variable B_i where $\langle B_i, \textsc{True}\rangle \in S$, then in any S-assignment which satisfies c, $B_i = \textsc{True}$. GAC propagation therefore removes $\langle B_i, \textsc{False}\rangle$ from S and the constraint is true for all assignments in the search state and therefore can have an empty support.

We now consider the search states where $\{\langle B_i, \textsc{False}\rangle \mid i \in \{1...n\}\} \subseteq S$ and $|\{\langle B_i, \textsc{True}\rangle \mid i \in \{1...n\}\} \cap S| \geq 2$. The minimal supports for these search states are of the form $\{\langle B_i, \textsc{True}\rangle, \langle B_j, \textsc{True}\rangle\}$ for $i \neq j$, where these supports are contained in S. To see this, if we reduced our search state to contain only one TRUE literal, then the FALSE must be removed for that variable, as discussed above. As long as two TRUE literals exist, propagation cannot occur as we do not know which variable will be assigned to a solution. Note, the constraint may become entailed by some FALSE literal being removed by search or another constraint, but this does not effect the correctness of the support.

Example 7. Consider the **parity** constraint c defined as $(\sum_{i=1...n} b_i) \bmod 2 = 0$ where $b_1 \ldots b_n$ have domain $\{0, 1\}$, and a search state S where $\mathrm{GAC}_c(S) = S$.

If S contains one literal for every variable, then the assignment represented by these literals must satisfy c. If S contained two literals for one variable b_i, then there are exactly two S-assignments, which differ only on their assignment to b_i. Exactly one of these assignments will satisfy c and therefore one of the two literals of b_i in S should be removed by GAC propagation.

Therefore, we can assume S contains both literals of at least 2 variables. We will show in this case, any support U for c must contain both literals of two variables. From Theorem 2, we must be able to extend any S-assignment to any single variable b_i and all variables not in U, using only literals from U, an S-assignment of $\sigma(c)$ which satisfies c. As long as there exists some b_j (with $i \neq j$) where both literals of b_j are in U, we can always satisfy this condition, as taking any S-assignment which does not satisfy c and changing the assignment to b_j will give an assignment which satisfies c. The final case is supporting the variable b_j. By the same

argument this requires another variable B_k which has both literals in the support.

Example 8. Consider the **alldifferent** constraint on an array of variables $X_1, \ldots, X_n$, of domain size $\{1, \ldots, n\}$. Consider a single literal $l = \langle X_i, j\rangle$. In any satisfying assignment, no other variable can be assigned j, and therefore given a search state S which contains $\langle X_k, j\rangle$ for some $k \neq i$, a support for S must contain some literal $\langle X_k, l\rangle$, for $l \neq j$. If S contains $\langle X_i, l\rangle$, then the support must also contain $X_k, m\rangle$, for some $l \neq m$. This means that for many domains, supports must contain at least $2 \times n$ literals (they may have to contain more). The support used by Nightingale et al. (Nightingale 2011) contained $3 \times n$ literals. The required number of literals is between 2 and 3 times the number of unassigned variables, and varies by domain.

Characterising supports for Assign propagators is much simpler.

Theorem 3. Given a constraint c and a search state S, then a set of literals $U \subseteq S$ is a support in S for Assign_c if and only if one of the following is true:

- U contains more than one literal for at least one variable in $\sigma(c)$.
- U contains at most one literal for each variable in $\sigma(c)$ and any S-assignment to $\sigma(c)$ which contains U satisfies c.

Proof. Assign_c only removes literals when the current search state consists of exactly one literal for each variable v in $\sigma(c)$, and the assignment represented by these literals does not satisfy c. If U contains more than one literal for any variable in $\sigma(c)$, then in any search state S' with $U \subseteq S' \subseteq S$, $\text{Assign}_c(S') = S'$.

If U contains at most one literal for each variable in $\sigma(c)$, then for U to be a correct support there must be no assignment to the variables in $\sigma(c)$ that contains the literals in U and does not satisfy c, as the search state S' containing only this assignment would satisfy $U \subseteq S' \subseteq S$, but $\text{Assign}_c(S') = \emptyset$. Otherwise, As propagators do not have to consider search states where any variable has no literals, any search state S' where $U \subseteq S' \subseteq S$ satisfies $\text{Assign}_c(S') = S'$. □

We can generalise the supports given in Example 7, which required 2 literals on 2 variables, to a much larger class of propagators. To do this we first need to define the Forward Checking propagator.

Definition 8. The **Forward Checking** propagator for a constraint c, FC_c, is defined on a search state S as follows:

1. If more than one variable in $\sigma(c)$ has more than one literal in S, then $\text{FC}_c(S) = S$.
2. If no variable in $\sigma(c)$ has more than one literal in S, then if the single S-assignment to $\sigma(c)$ does not satisfy c then $\text{FC}_c(S) = \{\}$, else $\text{FC}_c(S) = S$.
3. If exactly one variable $v \in \sigma(c)$ has more than one literal in S, then FC_c removes from S each literal $\langle v, i\rangle \in S$ where the unique S-assignment to $\sigma(c)$ containing $\langle v, i\rangle$ does not satisfy c.

Unlike GAC propagators, forward checking propagators always require small supports, as Theorem 4 shows.

Theorem 4. Consider a constraint c and a search state S. If there exists $x_1, x_2 \in \sigma(c)$ where S contains two literals for both x_1 and x_2, then any set of literals containing two literals from S for each of two different variables, is a correct support for FC_c. If S contains more than one literal from at most one variable in $\sigma(c)$, then the empty support is correct for FC_c.

Proof. The Forward Checking propagator cannot remove any literals until only one variable has more than one value. As long as two variables each have two values, this cannot occur. When only one variable is allowed more than one value, the propagator has already removed all the literals that can be removed, at the point when the previous deletion of a value from the domain of another variable in $\sigma(c)$ occurred. Any further reduction in the domains will not lead to further deletions by the propagator, so no support is required. □

An example of the forward checking support from Theorem 4 is given in Example 7. Unlike Theorem 3, Theorem 4 does not categorise all minimal sets of supports, only showing a size four support is always sufficient. We can use the supports from Theorem 4 to also support GAC propagators for constraints where $\text{FC}_c = \text{GAC}_c$. Definition 9 defines when a constraint is trivially fixable, which Theorem 5 shows is equivalent to the constraint satisfying $\text{FC}_c = \text{GAC}_c$. This means that for trivially fixable constraints we can find size 4 supports for GAC_c.

Definition 9. A constraint c is **trivially fixable** if, given any assignment to $\sigma(c)$ that is not allowed by c, changing the assignment of any variable to a different value creates an assignment that is allowed by c. Constraints with this property include **parity** (from Example 7); $\sum c_i * x_i \neq y$ where the constants c_i are not zero; and $\prod x_i \neq y$ as long as 0 is not in the domain of any x_i.

Theorem 5. A constraint c is trivially fixable if and only if $\text{GAC}_c = \text{FC}_c$.

Proof. First, let us consider a trivially fixable constraint, c. In a search state S where at least two variables are allowed more than one literal, then given any S-assignment A which contains a literal l, either A satisfies c, or we can change at least one assignment in A other than l to produce A', which will satisfy c by the definition of trivially fixable. Propagation can only occur when S contains more than one literal for exactly one variable, where FC_c removes the same set of literals GAC_c would remove. Therefore $\text{FC}_c = \text{GAC}_c$ for trivially fixable constraints.

For a constraint c that is not trivially fixable, we will construct a search state in which FC_c does not remove any literals, but GAC_c will. Since c is not trivially fixable, there must exist assignments A and A', which differ in the value of only one variable, say v, and neither of which satisfies c. Let v' be another variable that has more than one value: suppose its value in A and A' is d, and it has another value d'. Consider the search state S which consists of the literals in A and A' and the literal $\langle v', d'\rangle$. FC_c will not remove any literals

from S, as S contains more than one literal from two variables. However, the literal $\langle v', d\rangle$ cannot be extended to an S-assignment that satisfies c, because only the assignments A and A' contain it, and neither satisfies c. Hence, GAC_c will remove (at least) $\langle v', d\rangle$. Therefore, for constraints that are not trivially fixable, $\text{GAC}_c \neq \text{FC}_c$. □

The Minion (Gent, Jefferson, and Miguel 2006a) constraint solver uses this result to provide small supports for GAC propagation of trivially fixable constraints. This section has shown how to verify the correctness of a support, and showed an important class of constraints (trivially fixable) which allow very small supports. Finding other classes which allow small supports is important future work.

Complexity of finding and verifying supports

Finding and verifying the correctness of supports can be much more difficult than propagation. In this section we will demonstrate a propagator which can be run in polynomial time but where checking if a support of a given size exists is NP-hard. Further we will show a propagator which is NP-hard to execute, but Π_2^P-hard (Stockmeyer 1976) to check if a support is correct. First we will consider a variant of the 2-dimensional element constraint, given in Definition 10. The **element2equality** constraint is an unnatural constraint, designed specifically to easily prove finding supports can be difficult – the same results holds for GAC propagation of 2-dimensional element, but the proof is much longer, and more complicated, so must be ommited for space.

Definition 10. Given a matrix M of $n \times n$ variables and variables X, Y, R, A, B, where each variable has domain $\{1..n\}$, the **element2equality** constraint is the constraint $(M[X, Y] = R) \wedge (A = B)$.

The **simple** propagator for **element2equality** is the propagator which performs the following five steps, given a search state S:

1. For each literal $\langle A, i\rangle \notin S$, remove $\langle B, i\rangle$ and vice-versa.
2. For $\langle X, x\rangle \in S$, if there does $\nexists y \in \{1...n\}.\exists r \in \{1...n\}.\langle M[x, y], r\rangle \in S$, then remove $\langle X, x\rangle$ from S.
3. For $\langle Y, y\rangle \in S$, if there does $\nexists x \in \{1...n\}.\exists r \in \{1...n\}.\langle M[x, y], r\rangle \in S$, then remove $\langle Y, y\rangle$ from S.
4. For $\langle R, r\rangle \in S$, if there does $\nexists x \in \{1...n\}.\exists y \in \{1...n\}.\langle M[x, y], r\rangle \in S$, then remove $\langle R, r\rangle$ from S.
5. If S contains a single literal for each of A, B, X, Y and R, then consider the literals $\langle X, x\rangle$, $\langle Y, y\rangle$ and $\langle R, r\rangle$ in S for X, Y and R. If $\langle M[x, y], r\rangle \in S$, then do nothing, else return $\emptyset$.

The propagator described in Definition 10 can be executed in polynomial time, as its description can be directly turned into code. The last condition ensures that when all variables are assigned, then the constraint checks the resulting assignment satisfies the constraint.

The reason for the simple propagator for **element2equality** is given by Corollary 1, which proves that finding supports for this propagator is equivalent to tripartite perfect matching (Definition 11), proved to be NP-hard in (Karp 1972).

Definition 11. Given three disjoint sets A, B, C of size n, a **tripartite graph** on A, B, C is a set of tuples E where each element of E is of the form $\langle a, b, c\rangle$ where $a \in A, b \in B$ and $c \in C$. A **tripartite perfect matching** T for E on A, B and C satisfies $T \subseteq E, |T| = n$ and $\forall i \in A \cup B \cup C$, some tuple in T contains i.

Theorem 6 gives the form of supports of **element2equality** for a certain search states, which Corollary 1 will show match with finding tripartite graph matchings, and are thus NP-hard to find.

Theorem 6. Consider a matrix M of $n \times n$ variables and variables X, Y, R, A, B, each with domain $\{1..n\}$, and a search state S of these variables where X, Y, R, A and B have their initial domains. Then a correct support for the constraint **element2equality** on S must contain all the literals of both A and B, and a literal of the form $\langle M[x, y], r\rangle$ for each $x, y, r \in \{1...n\}$.

Proof. From Definition 10, if any literal from $\langle A, i\rangle$ is removed, then propagation will remove $\langle B, i\rangle$ and vice-versa. Therefore the support must contain all literals from both A and B. The second condition from Definition 10 requires the propagator remove $\langle X, x\rangle$ if there are no literals of the form $\langle M[x, y], r\rangle$ for some y and r. Therefore our support must contain at least one such literal (and similarly for Y and R).

This is all the support we require – if none of the literals are removed then the last condition from Definition 10 cannot trigger, as the domains of both A and B will be complete. □

Corollary 1. Given the **element2equality** constraint on domains of size n, using the propagator given in Definition 10, finding if there exists a support of size $n \times 3$ is NP-hard.

Proof. Consider a tripartite graph E on 3 sets of vertices A, B, C each of size n. We will transform the problem of finding a perfect matching for E into the problem of finding a support for **element2equality** of size $n \times 3$.

Assume that E contains at least one hyper edge for each vertex in A, B and C, else there is trivially no perfect matching. We will generate a search state S for **element2equality** of the form required by Theorem 6. The literals we use for matrix M will be those of the form $\langle M[x, y], r\rangle$ where $\langle x, y, r\rangle \in T$, and the complete domains for each of X, Y, R, A, B.

A subset of T which forms a tripartite prefect matching can be transformed into a correct support by mapping each element of T to a literal of M, and adding the literals of both A and B. Similarly a support for **element2equality** of size $n \times 3$ can be transformed into a tripartite perfect matching by taking the literals from M and taking the element of T which represents each one, as by Definition 10, this must contain a literal which contains each domain value of X, Y and R. □

We will now consider finding supports for NP-hard propagators. Examples of NP-hard propagators include symmetry breaking constraints (Jefferson et al. 2006) and bin packing constraints (Trick 2003).

Given a SAT instance s on variables $x_1, \ldots, x_n, y_1, \ldots, y_m$, the Π_2^P complexity class is

equivalent to the problem of checking if for all assignments to the x_i variables there exists an assignment to the y_i variables which satisfies s. This complexity class is above NP-complete in the polynomial hierarchy, for more details see (Stockmeyer 1976). Lemma 1 shows an NP-hard propagator where checking if a support is correct is Π_2^P-hard.

Lemma 1. Consider the class of constraints c_S, which check if a SAT instance S on variables $x_1, \ldots, x_n$ is satisfied. The propagator GAC_{c_S} is NP-hard, while checking if a support for GAC_{c_S} is correct is Π_2^P-hard.

Proof. Bessiere et al. show in (Bessière et al. 2004) that GAC propagating a constraint c is NP-hard if and only if the problem of checking if there exists a satisfying assignment to c within a search state is NP-hard. For the complete search state this is equivalent to solving S, so GAC_{c_S} is NP-hard.

Take a SAT instance S, and create a new instance S' by adding a new boolean variable x to the variables of S, and the variable x to every clause of S. S' is true for any assignment where x is true, and in assignments where x is false S' is true if and only if S would be true for the same assignment. Assuming S has at least one satisfying assignment, then the initial search state for c_S satisfies $GAC(S)$.

Now, consider a support which contains $\langle x, \mathrm{TRUE}\rangle$ and every literal for some subset A of the variables in S. Is this a correct support? Using Theorem 2, every literal in every variable other than x is supported by $\langle x, \mathrm{TRUE}\rangle$, and $\langle x, \mathrm{TRUE}\rangle$ is supported by any assignment containing it. Therefore the only literal we need to check for support is $\langle x, \mathrm{FALSE}\rangle$. This will be a correct support when, for all assignments to the variables not in A, there exists an assignment to the variables in A which satisfies S. This is the definition of Π_2^P-hard. □

This section has shown that finding small supports for a propagator can be harder than the propagator itself. This shows the importance of analysing classes of propagators to show when small supports can be found efficiently.

Search-Dependant Supports

In the previous sections, we have analysed finding a support for a propagator and a search state. During search we move between search states and eventually a literal from a support will be removed. The option considered thus far in this situation is to find a new support for the propagator. We call this system **dynamic supports**.

Another option, considered in this section, is to build a single static support at the start of search and use it throughout. This is a more limited but has the advantage that it can be implemented in solvers both quickly and efficiently. For example, the Choco3 constraint solver (Charles Prud'homme 2014) only supports static triggers, as triggers cannot be revoked during search. This section will show how to find static supports for constraints, and how they are often much larger than dynamic supports.

Generalised Supports

We begin by defining a static generalised support, which is correct in multiple parts of search, as it can contain literals which are not in the current search state. At any particular search state, to check if a static generalised support is valid, we simply ignore all literals not in this current search state.

Definition 12. A set U is a **static generalised support** for the propagator p_c in S if given any search state S' such that $p_c(S') = S'$ and $S' \subseteq S$, then $U \cap S'$ is a support for S'.

Using a single generalised support for the entire search is more limited than dynamic supports, because they are set at the beginning of search and then never change. For many propagators, static generalised supports will be very large, containing most or all of the literals in a given search state. In this section we will describe which literals can be omitted from a static generalised support for a GAC propagator.

Definition 13. Consider a search state S, a constraint c, and the search state $S_{\mathrm{GAC}} = \mathrm{GAC}_c(S)$. A literal $l = \langle v, d\rangle$ in S is **redundant** if, given any S-assignment containing l which satisfies c, the assignment generated by replacing l with any other assignment to v contained in S_{GAC} also satisfies c.

An example of the redundant literals of a constraint is given below.

Example 9. Consider the constraint $c = (x < y)$ on domains $x, y \in \{1, 2, 3, 4\}$. Then the domains of x and y after GAC_c has been applied are $x \in \{1, 2, 3\}, y \in \{2, 3, 4\}$. As there is no assignment which satisfies c containing $\langle x, 4\rangle$ and $\langle y, 1\rangle$, these two literals are trivially redundant. The only satisfying assignment containing $\langle x, 3\rangle$ is $\{\langle x, 3\rangle, \langle y, 4\rangle\}$, and in this assignment we could replace $\langle x, 3\rangle$ with $\langle x, 2\rangle$ or $\langle x, 1\rangle$, so $\langle x, 3\rangle$ is also redundant. Similarly, $\langle y, 2\rangle$ is redundant.

We will now show the importance of redundant literals. Theorem 7 shows that any literal which is not redundant must appear in any static generalised support. Corollary 2 uses this result to show that including all non-redundant literals produces any static generalised support.

Theorem 7. Given a constraint c, a static generalised support for the propagator GAC_c in a search state S must contain every non-redundant literal for c in S.

Proof. Consider a non-redundant literal $l = \langle v, d\rangle$ in S. We shall build a search state $S' \subseteq S$ which contains l and satisfies $\mathrm{GAC}_c(S') = S'$, and where $\mathrm{GAC}_c(S' - l) \neq S' - l$. This will prove that l must be in any static generalised support.

Define $T = \mathrm{GAC}_c(S)$ and consider a non-redundant literal $l = \langle v, d\rangle$ in S. By Definition 13, there is an S-assignment containing l that satisfies c, so $l \in T$. As l is non-redundant there exists an S-assignment A containing l which satisfies c, and another literal $l' = \langle v, d'\rangle \in T$, such that the S-assignment A', generated from A by replacing l by l', does not satisfy c.

By definition of GAC, as $l' \in T$ there exists S-assignments containing l' and satisfying c. Choose one such an S-assignment, B, which differs from A on the fewest variables (B may not be unique). B must differ from A for at least one variable other than v, as A' does not satisfy c.

Consider the search state U which contains exactly the literals in $A \cup B$. U is clearly contained in S, and also satisfies $\mathrm{GAC}_c(U) = U$, as every literal in U is contained in at least one of A and B, which are assignments satisfying c.

Now consider what would happen if l were removed from U. We know there is at least one literal other than l which occurs in A and not in B. Choose one and call it l_A. If l is deleted from U, running GAC_c must also cause l_A to be deleted. Suppose otherwise: then there must be an assignment in the remaining literals of U containing l_A and satisfying c. Such an assignment contains l_A, l' and other literals that are in A or B or both. Recall that B is constructed to be an S-assignment satisfying c that differs from A in as few places as possible. The new assignment cannot be exactly B as l_A is not in B, so it must be B with at least one literal replaced by the corresponding literal in A. But then the assignment differs from A in fewer places than B and so cannot be an assignment that satisfies c.

Therefore, any static generalised support for S must contain l, as any support for U must contain l. □

Corollary 2. The unique minimal static generalised support for the GAC propagator for a constraint c in a search state S is exactly the set of non-redundant literals for c in S.

Proof. Theorem 7 showed any static generalised support must contain all non-redundant literals, therefore we must show these literals are sufficient.

In any search state $S' \subseteq S$ where $\mathrm{GAC}(S') = S'$, consider a S'-assignment which satisfies c and a redundant literal $l = \langle v, d \rangle \in S'$. Removing l cannot cause any propagation, because either there is no other literal for v in S' (so search fails), or any S'-assignment A which satisfies c which is being used for support can be replaced by containing by another S'-assignment A' which replaces l with any other literal of v in S'. □

Theorem 7 and its corollary show that static generalised supports for GAC propagators can be very large. There are many constraints with no redundant literals, including **element**, **alldifferent**, **parity** and **gcc** (Nightingale 2011).

Static Generalised Supports and Propagator Strength

In Theorem 1, we showed that stronger propagators require larger supports. The same is not true for static generalised supports. To begin, we will consider the static generalised support for an Assign propagator.

Theorem 8. Consider a constraint c on variables $x_1, \ldots, x_n$. Then the minimal static generalised support for Assign_c consists of every literal l where there is at least one assignment not containing l which does not satisfy c.

Proof. Assign_c must perform propagation in any search state S which contains one literal for any variable, and the assignment represented by this search state does not satisfy c. Given such a search state, consider any search state S' generated by adding one l literal to S. When this literal is removed from S', it must trigger propagation, and therefore it must be in the static generalised support.

On the other hand, the removal of a literal l where every assignment not containing l satisfies c would not trigger propagation, as removing l could only leave a satisfying assignment. □

In general, the static generalised support for a constraint will consist of almost all literals. Lemma 2 shows that weaker propagators can require either smaller or larger supports – there is no simple ordering of the size of static generalised supports as there was for dynamic supports in the previous section.

Lemma 2. Consider the constraint c defined as $A = B$, where A and B have domain $\{1, \ldots, 5\}$. Then there exists a propagator p for c where p is weaker than GAC_c and stronger than Assign_c, yet p has a smaller static generalised support than either GAC_c or Assign_c.

Proof. Consider the propagator p which removes the literal $\langle A, i \rangle$ when $\langle B, i \rangle$ is removed (we omit for space a proof that this is a correct propagator). The static generalised support for this propagator is all literals in $\langle B, i \rangle$, as it is only the removal of these literals which causes p to remove literals for A. Theorems 2 and 8 show that the minimal static generalised support for both GAC_c and Assign_c require all literals from both A and B. □

This section has shown that static generalised supports for GAC propagators include almost all literals. This demonstrates the importance of dynamic supports – we cannot in general take advantage of the theory of supports without also being able to change these supports dynamically.

Conclusion

We have presented a theory of static and dynamic supports for the CSP. We have shown that some types of constraint have small sets of dynamic supports and in contrast that the static generalised support for constraints contains almost every literal. This shows that we should use dynamic triggers if we wish a small set of supports; although we have found some cases where even with dynamic triggers a large set of supports are required. Further, we have proven that the use of dynamic triggers in some constraints allow a small set of triggers to achieve GAC propagation, the strongest possible propagator for a constraint. We have also proven that all constraints can have a small set of supports to achieve both assignment and forward checking propagation. Finally, we have shown that proving a set of literals is a support is theoretically hard, in particular it can be NP-hard for propagators which run efficiently in polynomial time.

In future work, we hope to consider backtrack-stable supports (also known as watched literals) further. These are much more difficult to analyse as they require considering both past and future search states. However, they are used frequently in practice and have had almost no formal study. Also, some constraint solvers allow constraints to be triggered on other events rather than just literals, for example changes in the upper and lower bound of variables, or when a variable becomes assigned. Backtrackable dynamic triggers can trigger on changes to the bound of a variable by containing the largest or smallest literal in the domain, we can assure we catch a variable being assigned by adding two literals from its domain to the support. We intend to fully investigate for which types of supports and constraints these events are useful.

References

Bessière, C.; Hebrard, E.; Hnich, B.; and Walsh, T. 2004. The tractability of Global Constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, 716–720. Toronto, Canada: Springer-Verlag.

Bessiere, C. 2006. Constraint propagation. *Handbook of constraint programming* 29–83.

Boisberranger, J. D.; Gardy, D.; Lorca, X.; and Truchet, C. 2013. When is it worthwhile to propagate a constraint? A probabilistic analysis of alldifferent. In *Proceedings of the 10th Meeting on Analytic Algorithmics and Combinatorics, ANALCO 2013, New Orleans, Louisiana, USA, January 6, 2013*, 80–90.

Charles Prud'homme, Jean-Guillaume Fages, X. L. 2014. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

Gecode Team. 2006. Gecode: Generic constraint development environment. `http://www.gecode.org`. Available from `http://www.gecode.org`.

Gent, I. P.; Jefferson, C.; and Miguel, I. 2006a. Minion: A fast scalable constraint solver. In *Proceedings of ECAI 2006*, 98–102. IOS Press.

Gent, I. P.; Jefferson, C.; and Miguel, I. 2006b. Watched literals for constraint propagation in minion. In *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *LNCS*. Springer Berlin Heidelberg. 182–197.

Gent, I. P.; Miguel, I.; and Nightingale, P. 2008. Generalised Arc Consistency for the AllDifferent Constraint: An Empirical Survey. *Artificial Intelligence* 172(18):1973–2000.

Green, M. J., and Jefferson, C. 2008. Structural tractability of propagated constraints. In Stuckey, P. J., ed., *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *LNCS*. Springer Berlin Heidelberg. 372–386.

Jefferson, C.; Kelsey, T.; Linton, S.; and Petrie, K. 2006. GAPLex: Generalised static symmetry breaking. In *The CP 2006 Workshop on Symmetry and Constraint Satisfaction Problems (SymCon'06)*, 17–23.

Jefferson, C.; Moore, N. C. A.; Nightingale, P.; and Petrie, K. E. 2010. Implementing logical connectives in constraint programming. *Artif. Intell.* 174(16-17):1407–1429.

Jefferson, C. 2011. Quicklex-a case study in implementing constraints with dynamic triggers. *ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2011* 97.

Karp, R. 1972. Reducibility among combinatorial problems. In Miller, R., and Thatcher, J., eds., *Complexity of Computer Computations*. Plenum Press. 85–103.

Katriel, I. 2006. Expected-case analysis for delayed filtering. In Beck, J., and Smith, B., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3990 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 119–125.

Lagerkvist, M. Z., and Schulte, C. 2007. Advisors for incremental propagation. In Bessire, C., ed., *Thirteenth International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, 409–422. Providence, RI, USA: Springer-Verlag.

Lagerkvist, M., and Schulte, C. 2009. Propagator groups. In Gent, I., ed., *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 524–538.

Mackworth, A. K., and Freuder, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence* 25(1):65 – 74.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC 2001*, 530–535. ACM.

Nightingale, P.; Gent, I. P.; Jefferson, C.; and Miguel, I. 2013. Short and long supports for constraint propagation. *Journal Artificial Intelligence Research (JAIR)* 46:1–45.

Nightingale, P. 2011. The extended global cardinality constraint: An empirical survey. *Artificial Intelligence* 175(2):586 – 614.

Stockmeyer, L. J. 1976. The polynomial-time hierarchy. *Theoretical Computer Science* 3(1):1 – 22.

Trick, M. 2003. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research* 118(1-4):73–84.