# Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization

James Bailey and Peter J. Stuckey

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne 3010, Australia

**Abstract.** An unsatisfiable set of constraints is minimal if all its (strict) subsets are satisfiable. The task of type error diagnosis requires finding all minimal unsatisfiable subsets of a given set of constraints (representing an error), in order to generate the best explanation of the error. Similarly circuit error diagnosis requires finding all minimal unsatisfiable subsets in order to make minimal diagnoses. In this paper we present a new approach for efficiently determining all minimal unsatisfiable sets for any kind of constraints. Our approach makes use of the duality that exists between minimal unsatisfiable constraint sets and maximal satisfiable constraint sets. We show how to incrementally compute both these sets, using the fact that the complements of the maximal satisfiable constraint sets are the hitting sets of the minimal unsatisfiable constraint sets. We experimentally compare our technique to the best known method on a number of large type problems and show that considerable improvements in running time are obtained.

**Keywords:** Minimal unsatisfiable sets, constraint solving, hitting sets, hypergraph transversals.

## 1 Introduction

A set of constraints is unsatisfiable if it has no solution. An unsatisfiable set of constraints is minimal if all its (strict) subsets are satisfiable. A number of forms of error diagnosis, in particular type error diagnosis, require finding all minimal unsatisfiable subsets of a given set of constraints (representing an error), in order to generate the best explanation of the error.

There is a significant amount of work that deals with minimal unsatisfiable sets, particularly in the areas of explanation and intelligent backtracking (e.g. [4]) or nogood creation (e.g. [16]). However, the vast bulk of this work is only interested in finding a single minimal unsatisfiable set. This is usually achieved by relying on some kind of justification recording, and then postprocessing the recorded unsatisfiable set to eliminate unnecessary constraints. In many cases a non-minimal unsatisfiable set is used.

Our motivation for examining the problem of finding all minimal unsatisfiable subsets of a set of constraints arises from type error debugging. In Hindley-Milner type inference and checking, a program is mapped to a system of Herbrand constraints and a type error results when this system of Herbrand constraints is unsatisfiable. An explanation of the type error is given by a minimal unsatisfiable subset of the system of Herbrand constraints

*Example 1.* Consider the following fragment of Haskell code

```
f [] y = []
f (x:xs) y = if (x < y) then (f xs y) else xs
g xs y = 'z' > (f xs y)
```

that defines a function `f` which returns a list, and then erroneously compares the result of that function to character `'z'`. The Chameleon type debugging system [17, 11] finds a single minimal unsatisfiable set of constraints that causes the type error and underlines the associated the program fragments. If that minimal unsatisfiable set included the constraints posed by the base case in the definition of function `f`, the Chameleon system would show the following

```
f [] y = []
f (x:xs) y = if (x < y) (f xs y) else xs
g xs y = 'z' > (f xs y)
```

while if the minimal unsatisfiable set included the constraints posed by the recursive case in the definition of function `f`, the Chameleon system would instead show the following

```
f [] y = []
f (x : xs ) y = if (x < y) (f xs y) else xs
g xs y = 'z' > (f xs y)
```

One explanation may be easier to understand than another, for example if it involves fewer constraints. Hence, deriving all minimal unsatisfiable sets allows us to choose the "simplest" explanation.

Finding all minimal unsatisfiable sets of a system of constraints is a challenging problem because, if it is done naively, it involves examining every possible subset. Indeed in the worst case there may be an exponential number of answers. Most previous work has concentrated on its use in diagnosis of circuit errors. The best method we know of is given by García de la Banda *et al* [5], who presented a series of techniques that significantly improved on earlier approaches of [13] and [12].

This paper presents a completely new method for calculation of all minimal unsatisfiable constraint sets. We show that this problem is closely related to a problem from the area of data mining, concerned with enumerating interesting sets of frequent patterns. In particular, we show how an algorithm known as *Dualize and Advance* [10], which has previously been proposed for discovering collections of maximal frequent patterns in data mining, can be efficiently adapted to the constraint context, to jointly enumerate both all minimal unsatisfiable sets and all maximal satisfiable sets of constraints.

Interestingly, Dualize and Advance, although having good worst case complexity, does not seem to be a practical algorithm for finding maximal frequent patterns in data mining [19, 8], due to the large number of patterns required to be output. However, in the constraint context, the size of the output (i.e. the number of minimal unsatisfiable sets of constraints and the number of maximal satisfiable sets of constraints) is typically far smaller and we demonstrate its efficiency. Furthermore, we show how improvements in the procedure can be made by incorporation of information from the constraint graph. We experimentally compare our method with the best known available technique from [5] on a number of debugging problems containing hundreds of constraints and show that our new approach can result in significant savings in running time. A further advantage of our approach for more traditional circuit diagnosis problems is that a possible diagnosis of an error corresponds to the complement of a maximal satisfiable set of constraints [15]. In our method these are easy to generate from the calculated maximal satisfiable sets.

The outline of the remainder of this paper is as follows. We first give some background definitions in Section 2. Next, we examine the best previous approach to the problem we are aware of, that of García de la Banda *et al* [5] in Section 3. In Section 4, we describe the Dualize and Advance approach and how it can be adapted and optimised for the constraint context. In Section 5 we describe the results of experiments comparing the two approaches. Finally in Section 6 we conclude and discuss future work.

## 2 Background

Let us start by introducing the notation which will be used herein. A constraint domain $\mathcal{D}$ defines the set of possible values of variables. A *valuation* $\theta$, written $\{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\}$, $d_i \in \mathcal{D}, 1 \leq i \leq m$, maps each variable $v_i$ to a value $d_i$ in the domain.

A *constraint* $c$ is a relation on a tuple of variables $vars(c)$. Let $vars(c) = (v_{i_1}, \ldots, v_{i_n})$ then $c$ defines a subset $vals(c)$ of $D^n$. A valuation $\theta \equiv \{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\}$ is a *solution* of constraint $c$ if $(d_{i_1}, \ldots, d_{i_n}) \in vals(c)$.

A set of constraints $C$ is *satisfiable* iff there exists a solution $\theta$ of $C$. Otherwise it is *unsatisfiable*. We assume an algorithm $sat(C)$ which returns $true$ if $C$ is satisfiable and $false$ otherwise.

We will also be interested in *incremental* satisfaction algorithms. Incremental satisfiability checks process each of the constraints one at a time. Hence, to answer the question $sat(\{c_1, \ldots, c_n\})$ we compute the answers to questions $sat(\{c_1\})$, $sat(\{c_1, c_2\})$, ..., $sat(\{c_1, \ldots, c_{n-1}\})$ and finally $sat(\{c_1, \ldots, c_n\})$. We describe incremental satisfiability algorithms as a procedure $isat(c_n, state)$ which takes a new constraint $c_n$ and an internal state representing a set of constraints $\{c_1, \ldots, c_{n-1}\}$ and returns a pair $(result, state')$ where $result = sat(\{c_1, \ldots, c_n\})$ and $state'$ is a new internal state representing constraints $\{c_1, \ldots, c_n\}$.

Since we are focusing on debugging, we will use the Herbrand equation constraint domain $\mathcal{H}$. That is, equations over uninterpreted function symbols, such as the constraint arising in Hindley-Milner typing. The complexity of *sat* for this class of constraints is $O(n)$ where $n$ is the number of symbols in the constraint [14]. The complexity of $n$ calls to *isat*, $(true, s_1) = isat(c_1, true)$, $(true, s_2) = isat(c_2, s_1)$, ..., $(result, s_n) = isat(c_n, s_{n-1})$ is $O(nA^{-1}(n))$ where $A^{-1}(n)$ is the inverse Ackerman's function. The amortized incremental complexity of $isat(c, state)$ is thus effectively constant. We will use calls to *isat* as one measure for the complexity of our algorithms. For this purpose as call $sat(\{c_1, \dots, c_n\})$ is equivalent to $n$ calls to *isat*.

For a given problem, we define the constraint universe $U$ as the set which contains every possible constraint that can be considered. In a typing problem these are all the type constraints represented by the program to be typed, while in circuit diagnosis it is all the constraints defining the circuit and its inputs and outputs.

A constraint set $C$ is a *minimal unsatisfiable* constraint set if $C$ is unsatisfiable and each $C' \subset C$ is satisfiable. A constraint set $C$ is a *maximal satisfiable* constraint set if $C$ is satisfiable and each $C' \supset C$ (where $C' \subseteq U$) is unsatisfiable. For a set of constraints $S$, we define its complement to be $\overline{S} = U - S$. Let $\mathbf{S} = \{S_1, S_2, \dots, S_k\}$ be a set of constraint sets (i.e. each of $S_1$, $S_2$ etc is a set of constraints). We define $\overline{\mathbf{S}}$, the complement of $\mathbf{S}$, to be the set of complements of each of the constraint sets. i.e. $\overline{\mathbf{S}} = \{\overline{S}_1, \overline{S}_2, \dots, \overline{S}_k\}$.

Let $\mathbf{A} = \{A_1, A_2, \dots, A_n\}$ be a set of constraint sets. We say a set $P \subseteq U$ is a *hitting set* of $\mathbf{A}$ if $(P \cap A_1 \neq \emptyset) \wedge (P \cap A_2 \neq \emptyset) \wedge \dots \wedge (P \cap A_n \neq \emptyset)$. We say that $P$ is a *minimal hitting set* of $\mathbf{A}$, if $P$ is a hitting set of $\mathbf{A}$ and each $S \subset P$ is not a hitting set of $\mathbf{A}$. We define $\mathcal{HST}(\mathbf{A})$ to be the set of all the minimal hitting sets of $\mathbf{A}$. The cross product of two sets of set $\mathbf{A} = \{A_1, \dots A_n\}$ and $\mathbf{B} = \{B_1, \dots, B_m\}$ is denoted as $\mathbf{A} \otimes \mathbf{B} = \{A_i \cup B_j \mid i \leq n, j \leq m\}$.

## 3 Best Previous Approach

The best previous approach we are aware of for finding all minimal unsatisfiable subsets of a constraint set is from García de la Banda *et al* [5], who extended approaches by [13] and [12]. Essentially all these approaches rely on enumerating all possible subsets of the constraints and checking which are unsatisfiable but all of whose subsets are satisfiable.

The code for min_unsat shown below gives the core. The call min_unsat($\emptyset$,$U$,$\emptyset$) it finds all minimal unsatisfiable subsets of $U$. The first argument is used to avoid repeatedly examining the same subset. The call min_unsat($D$, $P$, $\mathbf{A}$) traverses of all subsets of the set $D \cup P$ which include $D$, i.e. $\{D \cup P' \mid P' \subseteq P\}$. $D$ refers to definite elements, ones which must appear in all subsequent subsets and $P$ refers to possible elements, ones which may appear in subsequent subsets. The argument $\mathbf{A}$ collects all the minimal unsatisfiable subsets found so far. This call explores all subsets $\{D \cup P' \mid P' \subseteq P\}$ in an order such that all subsets of $D \cup P$ are explored before the **while** loop in the call min_unsat($D$,$P$,$\mathbf{A}$) finishes.

When a satisfiable subset is found then the algorithm need not look at its further subsets. If an unsatisfiable set is found then it is added to the collection $\mathbf{A}$ after all its subsets have been examined, unless there is already a subset of it in $\mathbf{A}$. The code returns the set of minimal unsatisfiable subsets found.

```
min_unsat($D$, $P$, $\mathbf{A}$)
    if ($sat(D \cup P)$) return $\mathbf{A}$
    while ($\exists c \in P$)
        $P := P - \{c\}$
        $\mathbf{A} :=$ min_unsat($D$, $P$, $\mathbf{A}$)
        $D := D \cup \{c\}$
    endwhile
    if ($\neg\exists A \in \mathbf{A}$ such that $A \subset D$) $\mathbf{A} := \mathbf{A} \cup \{D\}$
    return $\mathbf{A}$
```

This simple approach is improved in [5] by (a) detecting constraints that are present in all minimal unsatisfiable subsets by preprocessing, (b) taking into account constraints that must always be satisfiable once other constraints are not present, (c) using reasoning about independence of constraints to reduce the number of subsets examined, and most importantly (d) using incremental constraint solving to select which elements $c$ to select first in the **while** loop. The last modification is the most important in terms of reducing the amount of satisfaction checking and subsets examined.

Essentially by performing the satisfiability check $sat(D \cup P)$ incrementally we find the first constraint $c_i$ where $D \cup \{c_1, \ldots, c_{i-1}\}$ is satisfiable and $D \cup \{c_1, \ldots, c_i\}$ is not. This guarantees that $c_i$ appears in some minimal unsatisfiable set. By choosing $c = c_i$ in the while loop we (hopefully) quickly find large satisfiable subsets thus reducing the search.

## 4 Dualization Approach

We now describe our new approach for determining all the minimal unsatisfiable sets of constraints. It is based on a technique that has been proposed in the area of data mining, called Dualize and Advance [10], for discovery of interesting patterns in a database. Other similar algorithms exist from work in hypergraph transversals [3].

The key idea is that for a given constraint universe $U$, there exists a relationship between the minimal unsatisfiable sets of constraints and the maximal satisfiable sets of constraints. In particular, suppose we have a set $\mathbf{G}$ of satisfiable constraint sets, then $\mathcal{HST}(\overline{\mathbf{G}})$ is the collection of the smallest sets which are not contained in any set from $\mathbf{G}$. If we let $\mathbf{G}$ be the collection of all the maximal satisfiable constraint sets, then $\mathcal{HST}(\overline{\mathbf{G}})$ is the collection of all the smallest sets that are not contained in any maximal satisfiable set (i.e. the minimal unsatisfiable constraint sets). Furthermore, if $\mathbf{G}$ is a collection of some, but not all the maximal satisfiable constraint sets, then $\mathcal{HST}(\overline{\mathbf{G}})$ must contain at least one set which is satisfiable and is not contained in any set in $\mathbf{G}$ (see [10] for proof).

**Example 1** *Suppose for universe* $U = \{c_1, c_2, c_3, c_4\}$ *the maximal satisfiable sets of constraints are* $\mathbf{G} = \{\{c_3\}, \{c_4\}, \{c_2\}\}$. *Then the complements sets are* $\overline{\mathbf{G}} = \{\{c_1, c_2, c_3\}, \{c_1, c_3, c_4\}, \{c_1, c_2, c_4\}\}$ *and the minimal unsatisfiable sets are* $\mathcal{HST}(\overline{\mathbf{G}}) = \{\{c_1\}, \{c_2, c_4\}, \{c_2, c_3\}, \{c_3, c_4\}\}$.

We now present the algorithm for jointly generating both the minimal unsatisfiable sets and the maximal satisfiable sets. Although the stated purpose of our work is to find the minimal unsatisfiable sets, it is worth noting that the maximal satisfiable sets are also useful. In particular, if there are several possible error explanations, then the constraints which appear in the most maximal satisfiable sets are least likely to be in error. Similarly for circuit diagnosis, the minimal diagnoses are the complements of the maximal satisfiable sets [15].

The dualize and advance algorithm daa_min_unsat is given in Figure 1. The explanation is as follows. The algorithm maintains a number of variables: $X$ is a satisfiable set, which is grown into a maximal satisfiable set $M$ by the grow procedure which simply adds new constraints that do not cause unsatisfiability; $\mathbf{A}$ is the set of minimal unsatisfiable subsets currently found; $\mathbf{X}$ is the complements of the maximal satisfiable sets currently found; and $\mathbf{N}$ are the hitting sets for $\mathbf{X}$ which are the candidates for minimal unsatisfiable subsets.

Initially all the set of minimal unsatisfiable sets and complements of maximal satisfiable sets are empty. The $X$ variable is set to $\emptyset$. In the **repeat** loop, the algorithm, repeatedly grows $M$ to a maximal satisfiable subset, adds its complement to $\mathbf{X}$ and calculates the new hitting sets for $\mathbf{X}$. This gives the candidates $\mathbf{N}$ for minimal unsatisfiable subsets.

For each of these not already recognised as a minimal unsatisfiable subset we check satisfiability. If the set $S$ is satisfiable then it is the starting point for a new maximal satisfiable set, and we break the **for** loop and continue. Otherwise $S$ is added to the minimal unsatisfiable subsets $\mathbf{A}$. When we find no satisfiable $S$ then we have discovered all minimal unsatisfiable subsets.

**Example 2** *We trace the behaviour of the algorithm daa_min_unsat using Example 1, where the minimal unsatisfiable sets are* $\mathbf{G} = \{\{c_1\}, \{c_2, c_4\}, \{c_2, c_3\}, \{c_3, c_4\}\}$ *and the maximal satisfiable sets are* $\{\{c_4\}, \{c_3\}, \{c_2\}\}$. *We show the values of key variables just before the* ***for*** *loop for each iteration of the* ***repeat*** *loop.*

| *Iter.* | $M$ | $\mathbf{X}$ | $\mathbf{N} = \mathcal{HST}(\mathbf{X})$ |
|---|---|---|---|
| *1* | $\{c_2\}$ | $\{\{c_1, c_3, c_4\}\}$ | $\{\{c_1\}, \{c_3\}, \{c_4\}\}$ |
| *2* | $\{c_3\}$ | $\{\{c_1, c_3, c_4\}\}, \{c_1, c_2, c_4\}\}$ | $\{\{c_4\}, \{c_1\}, \{c_2, c_3\}\}$ |
| *3* | $\{c_4\}$ | $\{\{c_1, c_3, c_4\}, \{c_1, c_2, c_4\}, \{c_1, c_2, c_3\}\}$ | $\{\{c_1\}, \{c_2, c_3\}, \{c_2, c_4\}, \{c_3, c_4\}\}$ |

Each iteration produces a new complement of a maximum satisfiable set. Once all maximal satisfiable sets have been found, the **repeat** loop terminates.

## 4.1 Determining the Hitting sets $\mathcal{HST}$

A core part of the procedure is the calculation of the hitting sets on each iteration of the while loop. There are many possible methods for computing hitting

```
daa_min_unsat($U$)
    $\mathbf{A} := \emptyset$
    $\mathbf{X} := \emptyset$
    $X := \emptyset$
    repeat
        $M :=$ grow($X$,$U$);
        $\mathbf{X} := \mathbf{X} \cup \{U - M\}$
        $\mathbf{N} := \mathcal{HST}(\mathbf{X})$
        $X := \emptyset$
        for ($S \in \mathbf{N} - \mathbf{A}$)
            if ($sat(S)$)
                $X := S$
                break
            else $\mathbf{A} := \mathbf{A} \cup \{S\}$
        endfor
    until ($X = \emptyset$)
    return ($\mathbf{A}$)

grow($S$,$U$)
    for ($c \in U - S$)
        if ($sat(S \cup \{c\})$) $S := S \cup \{c\}$
    endfor
    return($S$)
```

**Fig. 1.** Dualize and advance algorithm for finding minimal unsatisfiable sets.

sets. This problem is also known as the *hypergraph transversal problem*. We use a method first described by Berge [2], since it is simple to implement and behaves reasonably efficiently. More complex techniques do exist though which have better worst case complexity (see [7]) or are better in practice for large problems (see [1]). The basic idea of the Berge algorithm is that to compute the hitting sets of a set $G$, the sets contained in $G$ are first ordered, and then partial cross products of these sets are computed, with the output being minimised at each step.

Let $\mathbf{G} = \{S_1, S_2, \ldots, S_k\}$ and define $\mathbf{G}_i = \{S_1, \ldots, S_i\}$. Then $\mathcal{HST}(\mathbf{G}_i)$ is given by the formulas

$$\mathcal{HST}(\mathbf{G}_1) = \{\{c\} \mid c \in S_1\}$$
$$\mathcal{HST}(\mathbf{G}_2) = \mathcal{HST}(\mathbf{G}_1 \cup \{S_2\}) = Min(\mathcal{HST}(\mathbf{G}_1) \otimes \{\{c\} \mid c \in S_2\})$$
$$\ldots$$
$$\mathcal{HST}(\mathbf{G}_i) = \mathcal{HST}(\mathbf{G}_{i-1} \cup \{S_i\}) = Min(\mathcal{HST}(\mathbf{G}_{i-1}) \otimes \{\{c\} \mid c \in S_i\})$$

where $Min(\mathbf{G})$ is the set $\mathbf{G}$ with all non-minimal subsets removed.

$$Min(\mathbf{G}) = \{S \mid S \in \mathbf{G} \wedge (\forall T \in \mathbf{G}\ (T \subseteq S) \Rightarrow (T = S)\}.$$

## 4.2 Incremental Hitting set Calculation

Looking more closely at the procedure for minimal hitting set calculation, we can see that it is incremental in nature—the hitting sets $\mathcal{HST}(\mathbf{G})$ of a set $\mathbf{G}$ are computed by considering each set from $\mathbf{G}$ in turn and calculating the partial hitting sets. Thus, by remembering the partial hitting sets $\mathcal{HST}(\mathbf{G}_1)$, $\mathcal{HST}(\mathbf{G}_2)$, etc, we can incrementally calculate the new hitting sets of $\mathbf{X}$ when a new set of constraints is added. Hence we can replace the line

$\mathbf{N} := \mathcal{HST}(\mathbf{X})$

by

$\mathbf{N} := Min(\mathbf{N} \otimes \{\{c\} \mid c \in U - M\})$

## 4.3 Complexity of the the algorithm

The complexity of the algorithm is as follows. The number of iterations of the **repeat** loop is equal to the number of maximal satisfiable sets, since in each iteration we find one more $M$. In each iteration we call grow once which costs at most $|U|$ incremental calls to *sat*, for Herbrand equations the cost is thus $O(|U|)$ overall. Each minimal unsatisfiable set is found within the **for** loop of daa_min_unsat and needs no further processing once it has been found to be unsatisfiable. Each minimal unsatisfiable set requires at most $|U|$ incremental calls to *sat*, thus again $O(|U|)$ overall for Herbrand equations. If $\mathbf{A}$ is the collection of all the minimal unsatisfiable sets and $\mathbf{X}$ is the collection of all the complements of maximal satisfiable sets, then overall the complexity (using the optimisation from section 4.2) is $O(|\mathbf{A}| \times |U| + |\mathbf{X}| \times |U| + cost(\mathcal{HST}(\mathbf{X})))$.

The core part of the cost is the calculation of the hitting sets $\mathcal{HST}(\mathbf{X})$, done incrementally. In general, the number of hitting sets (and thus the size of $\mathbf{X}$) can be exponential in $|U|$. Also, as we will show shortly (Example 3), the number of partial hitting sets may also be exponential, even when the size of $\mathbf{X}$ is polynomial. The addition of a new set $S_i$ to $\mathbf{G}$, can either increase the number of minimal hitting sets by a factor of $|S_i|$, or cause a superpolynomial decrease in the number of minimal hitting sets [18]. The exact complexity of the Berge algorithm is not yet well understood, but an upper bound for the $cost(\mathcal{HST}(\mathbf{X}))$ is $O(2^{|U|})$.

## 4.4 Optimisation of Hitting Set Computation Using the Constraint Graph

The order in which the sets of $\mathbf{G}$ are considered when computing the hitting sets can have a significant impact on the running time of the hitting set calculation. This is because the size of the partial hitting sets can blow up for certain orderings. An example (based on one from [6]) is:

**Example 3** *Let* $\mathbf{G} = \{\{c_i, c_j\} \mid i \leq 10, j \leq 10\}$ *Suppose we order the sets of* $\mathbf{G}$ *to be* $\{\{c_1, c_2\}, \{c_3, c_4\}, \{c_5, c_6\}, \{c_7, c_8\}, \{c_9, c_{10}\}, \ldots\}\}$*. Then* $|\mathcal{HST}(\mathbf{G}_5)| = 2^5$*, whereas if they are ordered as* $\{\{c_1, c_2\}, \{c_2, c_3\}, \{c_1, c_3\}, \{c_1, c_4\}, \{c_2, c_4\}, \ldots\}$ *then* $|\mathcal{HST}(\mathbf{G}_5)| = 3$*. In other words, a blowup of the intermediate results occurs for the first ordering, but not the second.*

To address this problem, a natural heuristic to use is that the sets contained in $\mathbf{X}$ should be ordered in increasing cardinality, to minimise the number of partial hitting sets. However, we do not have direct information about the cardinality of the next $M$ to be generated.

A heuristic to try and achieve this is as follows: When maximising a set in the grow procedure, we should add constraints in an order that 'maximises' the number of constraints in the final grown $M$ set. Therefore, we should add the constraints most likely to cause unsatisfiability last of all. To identify such constraints, we use a constraint graph to help identify a global ordering of all the constraints in the universe, with constraints likely to cause unsatisfiability occurring at the end of the ordering and constraints not likely to cause unsatisfiability occurring at the start of the ordering.

Given a set of constraints $U$, the *constraint graph* $g(U)$ is a graph where each vertex in the graph corresponds to one of the constraints and there is an edge between two vertices $c_1$ and $c_2$ iff there exists a $v$ such that $v \in vars(c_1)$ and $v \in vars(c_2)$. We estimate the centre of the constraint graph (the vertex will the least maximal distance from all other nodes) and then order vertices according to their distance from the centre. Constraints closest to the centre are at the beginning of the ordering, since these are expected to participate in the most minimal unsatisfiable subsets and those furthest away from the centre are at the end of the ordering.

## 4.5 Discussion of the Algorithm

As mentioned, our hitting set algorithm is based on the Dualize and Advance algorithm for mining interesting patterns (sets of items) in a database [10]. Work in [19] also presented a practical implementation of Dualize and advance for data mining. Our approach differs from both these works in a number of ways

- The context is constraints and not sets of items.
- The size of the output in data mining problems (number of maximally satisfiable sets and number of minimal unsatisfiable sets) is huge, this means that Dualize and Advance is not practical for data mining requirements [8, 19]. However, in the constraint scenario the number of minimal unsatisfiable sets is likely to be small, since the scenario is type error debugging. We are not aware of any previous work where Dualize and Advance has been shown to be efficient for an important practical problem.
- Knowledge of the constraint graph can be used as a means for improving the algorithm.

Dualize and Advance is quite similar to Reiter's approach for model based diagnosis ([15]), which uses the computation of hitting sets to relate conflict sets (similar to unsatisfiable sets) and diagnoses (complements of maximum satisfiable sets). The key difference is that Reiter's approach uses hitting set calculation to obtain each new minimum unsatisfiable set, whereas Dualize and Advance uses hitting set calculation to obtain each new maximum satisfiable set. Dualize and Advance has the important advantage that a satisfiable set can be grown into a maximum satisfiable set using an incremental solver. Reiter's technique requires a minimal unsatisfiable subset to be obtained from a larger unsatisfiable set by removal of constraints and there isn't the same opportunity for incremental solver use. Reiter's method also requires the costly maintenance of a tree structure for computing the hitting sets.

## 5 Experimental Evaluation

In order to investigate the benefits of our technique, we have implemented a prototype system in SICStus Prolog. The evaluation uses a number of benchmark problems arising from type error debugging. These are taken from sets of constraints generated by the Chameleon system [17] for debugging Haskell programs and use the efficient satisfiability procedure for solving Herbrand equations provided by SICStus Prolog. Figure 2 shows the characteristics of these benchmarks: the number of constraints $|U|$, the number of minimal unsatisfiable subsets $|\mathbf{A}|$ (and in brackets the average size of each minimal unsatisfiable subset) and the number of maximal satisfiable subsets $|\mathbf{X}|$ (and in brackets the average size). Note that $|\mathbf{X}|$ is the number of complements of the maximal satisfiable sets, which is equal to the number of maximal satisfiable sets. These benchmark problems were chosen due to their challenging size and each has a constraint universe of size 72 or more constraints. Naïvely each problem then requires considering at least $2^{72}$ subsets.

The algorithms from Sections 3 and 4 were coded in SICStus Prolog. The experiments were run on a Dell PowerEdge 2500 with Intel PIII, 1GHz CPU and 2 GB memory. All times are measured in seconds. We compare against two versions of the algorithms from [5]. The first 3.648 (using the terminology of that paper) performs (a) preprocessing to detect constraints in all unsatisfiable subsets, (b) eliminates constraints which are always satisfiable after other constraints are deleted, (c) breaks constraints up that are independent and (d) uses the incremental search approach. The second 3.8 simply combines (a) and (d). The first version is the method that (in general) examines the fewest subsets, while the second is the one that (in general) performs the fewest *isat* checks.

Figure 3 shows the running times for four different algorithms. The first two sets of times are those from the system [5] with the parameter sets 3.648 and 3.8. The second two sets of running times are for the Dualize and Advance algorithm. DAA.1 uses the basic algorithm with the optimisations from 4.2. DAA.2 uses the basic algorithm with the optimisations from 4.2 and 4.4

| Benchmark | $\lvert U \rvert$ | $\lvert \mathbf{A} \rvert$ | $\lvert \mathbf{X} \rvert$ |
|---|---|---|---|
| `const` | 72 | 6 (37) | 88 (70) |
| `rotate` | 81 | 8 (65) | 68 (80) |
| `filter` | 98 | 12 (24) | 5427 (95) |
| `drop` | 156 | 11 (62) | 599 (152) |
| `rot13` | 159 | 9 (10) | 376379 (154) |
| `permute` | 239 | 16 (77) | 120 (237) |
| `plot` | 448 | 10 (15) | 300 (445) |
| `diff` | 610 | 4 (46) | 46 (609) |
| `msort` | 1016 | 4 (34) | 2699 (1013) |

**Fig. 2.** Type error benchmark problems

| Benchmark | [5]3.648 | [5]3.8 | DAA.2 | DAA.1 |
|---|---|---|---|---|
| `const` | 1.8 | 1.3 | 1 | 1 |
| `rotate` | 1.3 | 1.5 | 1 | 1 |
| `filter` | 16441 | 7505 | 381 | 644 |
| `rot13` | 1269 | 71089 | 27780 | 30807 |
| `drop` | 2152 | 3037 | 77 | 145 |
| `permute` | 296 | 57 | 14 | 7 |
| `plot` | 1932 | 1868 | 44 | 46 |
| `diff` | 387 | 11 | 14 | 15 |
| `msort` | 908543 | 91412 | 1430 | 1420 |

**Fig. 3.** Comparative running times to find all minimal unsatisfiable sets (Seconds)

Looking at Figure 3, we see that the hitting set algorithms are substantially faster in the majority of cases. The one exception is the `rot13` benchmark, where there are a huge number of maximally satisfiable sets. The structure of the `rot13` minimal unsatisfiable sets gives a clue as to why this may be so. There are nine of these, with several of them being quite different from one another (i.e. sharing few constraints). This may be because there is more than one independent type error in the `rot13` program. The `rot13` benchmark also illustrates the importance of independence optimizations for the approach of [5].

The ordering of the constraints only makes a slight difference in many cases (DAA.1 versus DAA.2), but in others it can reduce computation time by half.

In Figure 4 we show the number of calls to *isat* made by each of the algorithms. We see that the number of calls to *isat* is substantially reduced for the DAA algorithms versus the others. Both versions of DAA make exactly the same number of *isat* checks. Observe that the only difference in running time between DAA.2 and DAA.1 is the time taken for (incremental) hitting set calculation.

Looking at Figure 3, it is clear that for some of the problems, the amount of time taken, even for the best hitting set algorithm (DAA.2) may still be too high to be useful for interactive debugging. One strategy to cope with this is to

| Benchmark | [5]3.648 | [5]3.8 | DAA |
|---|---:|---:|---:|
| `const` | 7039 | 147945 | 6567 |
| `rotate` | 6934 | 81440 | 6028 |
| `filter` | 1956330 | 20692654 | 532208 |
| `rot13` | 86421 | 137706583 | 59844542 |
| `drop` | 118046 | 5154734 | 94041 |
| `permute` | 56464 | 106335 | 30119 |
| `plot` | 1649205 | 1649205 | 135200 |
| `diff` | 179685 | 180864 | 28856 |
| `msort` | 6890081 | 32858078 | 2743337 |

**Fig. 4.** Number of incremental satisfiability checks (calls to *isat*) to find all minimal unsatisfiable sets

| Benchmark | [5]3.648 | [5]3.8 | DAA.2 |
|---|---:|---:|---:|
| `const` | 1.5 | 0.2 | 0.8 |
| `rotate` | 2.6 | 0.2 | 1.1 |
| `filter` | 5.4 | 0.7 | 0.2 |
| `rot13` | 22.8 | 2.5 | 0.4 |
| `drop` | 41 | 4.9 | 23.6 |
| `permute` | 60.5 | 4.3 | 12.1 |
| `plot` | 158 | 8.7 | 41.9 |
| `diff` | 276 | 6.6 | 13.5 |
| `msort` | 4889 | 146 | 36.7 |

**Fig. 5.** Comparative running times to find one minimal unsatisfiable set (Seconds)

provide the user with each minimal unsatisfiable set as soon as it is found, rather than waiting until all have been computed. This way the user may begin trying to discover the source of the error earlier. Figure 5 shows the amount of time taken for each algorithm to output the first minimal unsatisfiable set found. We see that the amount of time taken is generally more acceptable for interactive use and indeed the non hitting set algorithm [5]3.8 is usually the fastest. A possible technique would thus be to run [5]3.8 in parallel with DAA.2 and stop [5]3.8 after finding the first minimal unsatisfiable set. This would provide the user with one minimal unsatisfiable set quickly, but would still be likely compute the entire collection of minimal unsatisfiable in an acceptable total elapsed time.

## 6 Conclusions and Future Work

Finding all minimal unsatisfiable sets is a challenging problem because it implicitly involves considering each possible subset of a given set of constraints. In this paper we investigated how to reduce as much as possible the number of constraints sets that need to be examined. We presented a new method which builds upon related work in data mining and showed it to be superior to the best known previous method.

A promising direction for future work is to investigate the tradeoffs between using the hitting set approach and that of [5] and see if a hybrid technique combining the advantages of both can be developed.

**Acknowledgements**

We would like to thank Kathryn Francis for assistance with the implementation and experimental evaluation, and Jeremy Wazny for providing the type error benchmark problems.

## References

1. J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 485–488, 2003.
2. C. Berge. *Hypergraphs, North Holland Mathematical Library*, volume 45. Elsevier Science Publishers B.V (North-Holland), 1989.
3. E. Boros, G. Gurvich, L. Khachiyan, and K. Makino. Dual bounded generating problems: Partial and multiple transversals of a hypergraph. *SIAM Journal on Computing*, 30(6):2036–2050, 2000.
4. B. Davey, N. Boland, and P.J. Stuckey. Efficient intelligent backtracking using linear programming. *INFORMS Journal of Computing*, 14(4):373–386, 2002.
5. Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, 2003.
6. T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.
7. Michael L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.
8. B. Goethals and M. Zaki. Advances in frequent itemset mining implementations: Introduction to FIMI03. In *[9]*.
9. Bart Goethals and Mohammed Javeed Zaki, editors. *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, FIMI'03*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.
10. D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.
11. C. Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Proc. of ESOP'03*, LNCS, pages 284–301. Springer-Verlag, 2003.
12. B. Han and S-J. Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(2):281–286, 1999.
13. A. Hou. A theory of measurement in diagnosis from first principles. *Artificial Intelligence*, 65:281–328, 1994.

14. M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
15. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.
16. J. Silva and K Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceeding of ICCAD'96*, pages 220–228, 1996.
17. M. Sulzmann and J. Wazny. Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.
18. K. Takata. On the Sequential Method for Listing Minimal Hitting Sets. In *Proceedings Workshop on Discrete Mathematics and Data Mining, 2nd SIAM International Conference on Data Mining*, 2002.
19. T. Uno and K. Satoh. Detailed description of an algorithm for enumeration of maximal frequent sets with irredundant dualization. In *[9]*.