Finding Relations Among Linear Constraints*

Jun Yan[1,2], Jian Zhang[1], and Zhongxing Xu[1,2]

[1] Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2] Graduate University, Chinese Academy of Sciences
{yanjun, zj, xzx}@ios.ac.cn

Abstract. In program analysis and verification, there are some constraints that have to be processed repeatedly. A possible way to speed up the processing is to find some relations among these constraints first. This paper studies the problem of finding Boolean relations among a set of linear numerical constraints. The relations can be represented by rules. It is believed that we can not generate all the rules in polynomial-time. A search based algorithm with some heuristics to speed up the search process is proposed. All the techniques are implemented in a tool called MALL which can generate the rules automatically. Experimental results with various examples show that our method can generate enough rules in acceptable time. Our method can also handle other types of constraints if proper numeric solvers are available.

1 Introduction

Constraints play an important role in various applications, and constraint solving has been an important research topic in Artificial Intelligence. A useful technique for constraint solving is to add some redundant constraints so as to improve the algorithms' efficiency [1, 2]. However, there is not much work on the systematic discovery of such constraints.

When we study constraint solving techniques, it is usually helpful if we take the form of constraints into account. This often depends on the application domain. One domain that is quite interesting to us is program analysis and verification. To analyse a program and generate test data for it, we may analyze the program's paths. For each path, we can derive a set of constraints whose solutions represent input data which force the program to be executed along that path [3]. Such a path-oriented method is often used in software testing, and it may also be used in infinite loop detection [4].

Generally speaking, the constraints encountered in program analysis and testing can be represented as Boolean combinations of arithmetic constraints [3, 5]. Here each constraint is a Boolean combination of primitive constraints, and a primitive constraint is a relational expression like $2x + 3y < 4$. In other words, a constraint is a Boolean formula, but each variable in the formula may stand for a relational expression. To solve such constraints, we developed a solver called BoNuS which combines Boolean satisfiability checking with linear programming.

* Supported in part by the National Science Foundation of China (grant No. 60125207 and 60421001).

In program analysis, we often need to solve many sets of constraints. They may contain some common primitive constraints. So if we can find some logic relationship (i.e. rules) between the primitive constraints (especially those occurring frequently) and add them as lemmas, the search space can be reduced.

Formal verification is also an important way to maintain program quality. Model checking is an effective verification method. But traditionally it is based on the propositional temporal logic. To scale it up to more real programs, one may use abstractio[6, 7]. It is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. A mapping between the actual data values in the system and a small set of abstract data values are given according to these relationships. In fact, since the abstract values are not always independent, these abstractions can be regarded as some rules deduced from the predicates. Therefore we need to find out some logic relations of the predicates before using abstraction.

In this paper, we try to employ a linear programming solver called `lp_solve` [8] to find all the logic relations among a set of linear arithmetic constraints automatically. We implemented a tool and used it to analyze how the attributes of a constraint set affect the number of rules. Since most of our techniques do not rely on any special characteristics of linear constraints, our method can be generalized to other types of constraints such as non-linear constraints if a proper solver is provided.

This paper is organized as follows. The next section will briefly introduce the problem of finding rules from numerical constraints and analyze its complexity. Then section 3 will present the main idea of our algorithm and some improving techniques. Experimental results and some analysis are given in Section 4. Then our approach is compared with some related works in Section 5, and some directions of future research are suggested in the last section.

2 The Problem and Its Complexity

2.1 Linear Arithmetic Constraints

In this paper, a numerical constraint is a *Linear Constraint* in the following form:

$$a_1x_1 + \ldots + a_nx_n \bowtie b$$

where a_i is a coefficient, x_i is a variable, and $\bowtie \in \{=, <, >, \leq, \geq, \neq\}$ is a relational operator. A conjuncion of linear constraints

$$\varphi : \begin{bmatrix} a_{11}x_1 & + \cdots + & a_{1n}x_n & \bowtie b_1 & \wedge \\ \vdots & \cdots & \vdots & \vdots & \vdots \\ a_{m1}x_1 & + \cdots + & a_{mn}x_n & \bowtie b_m & \end{bmatrix}$$

can be written concisely in matrix form as $A\boldsymbol{x} \bowtie \boldsymbol{b}$ where the bold $\boldsymbol{x}$ and $\boldsymbol{b}$ are n-dimensional and m-dimensional vectors, respectively.

Example 1. Here is an example of a set of numerical constraints.

```
C1 = (2x + y >= 4);
C2 = (x == y);
C3 = (x > 1).
```

We can use a Linear Programming (LP) package like `lp_solve` [8] to process linear constraints. In a linear programming problem, there can be a set of equations and standard inequalities (whose relational operators are $\geq$ and $\leq$). LP has been studied by many people and it can be solved by efficient algorithms [9].

Since most Linear Programming packages do not support the strict inequalities $\bowtie \in \{\neq, <, >\}$ (that are the negations of standard equations and inequalities respectively), we translate them into the standard form in the following way.

1. The expressions $Exp_1 < Exp_2$ and $Exp_1 > Exp_2$ can be replaced by $Exp_1 + \delta \leq Exp_2$ and $Exp_1 \geq \delta + Exp_2$ respectively, where δ is a very small positive number.
2. We use two expressions $A_1 = (Exp_1 < Exp_2)$ and $A_2 = (Exp_1 > Exp_2)$ to translate $A = (Exp_1 \neq Exp_2)$. This strict inequality will be replaced by $A = A_1 \vee A_2$. But LP solvers do not support the disjunction of constraints. So in the worst case, we may have to call `lp_solve` 2^n times if there are n such strict inequalities.

2.2 Constraints-Relation-Finding Problem

For the constraints in Example 1, we can easily find out that the following relation holds:

$$C1 \wedge C2 \rightarrow C3.$$

But if the number of constraints grows, the relations among the constraints can be quite complex and we need a method to obtain these rules automatically.

We can formalize the relations or rules as $A \rightarrow B$ where A and B are numerical constraints or their Boolean combinations. Note that

$$A \rightarrow B \text{ is tautology} \Leftrightarrow A \wedge \neg B \text{ is contradiction}$$

and we can transform $A \wedge \neg B$ into disjunctive normal form (DNF) such that each conjunctive clause is unsatisfiable. (A conjunctive clause is the conjunction of literals. It will be simply called a clause in this paper, unless stated otherwise.) So we only need to find out unsatisfiable clauses in the following form:

$$C_1 \wedge C_2 \ldots \wedge C_m$$

where m is the *length* of this clause. The literals $C_1, C_2, ...C_m$ represent the original numeric constraints or their negation.

It is natural to think of a clause as a set of literals. A clause that is a subset of another is called its *subclause*. For any clause ϕ, each constraint C_i may have one of the following 3 statuses: $C_i \in \phi$; $\neg C_i \in \phi$; neither. So the set of conjunctive clauses is the power set 3^C where C is the original constraint set. Our goal is to find out an unsatisfiable subset of this power set.

We define the Constraints-Relation-Finding (CRF) problem as follows:

Definition 1 (CRF). *Given a set of constraints $\mathcal{C} = \{C_i\}$, enumerate all unsatisfiable clauses $C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_m}$ where for all k, $1 \leq k \leq m$, $C_{j_k} \in \mathcal{C}$ or $\neg C_{j_k} \in \mathcal{C}$.*

It is well known that integer linear programming (recognizing the satisfiable instances) is NP-complete, so it is trivial that the converse, recognizing the set of unsatisfiable instances, is coNP-complete. So if we restrict the constraints to integer linear constraints, the subproblem is coNP-complete.

The minimal unsatisfiable sub-formula (MUS) is the following problem: Given a Boolean formula in conjunctive normal form with at most three occurrences of each variable, is it true that it is unsatisfiable, yet removing any clause renders it satisfiable? MUS is known to be DP-complete [10]. The subproblem of deciding if a sub-formula $C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_m}$ is unsatisfiable is also coNP-complete since it's the complementary problem of 3-SAT. If we solve the CRF problem for $\{C_i\}$, we solve the MUS problem for it: we only need to pick up the shortest sub-formula $C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_m}$ that is unsatisfiable. So the complexity of the CRF problem is at least the same as a DP-complete problem.

If we can solve the CRF problem by calling the constraint solver polynomial times, we can reduce the CRF problem to a coNP-complete problem in polynomial-time. But "DP contains a host of natural problems, whose membership in $NP \cup coNP$ is in serious doubt" [11]. So we believe that we can not solve the CRF problem by calling the constraint solver polynomial times. Otherwise, DP-complete problems would be as easy as coNP-complete problems.

From the above analysis, we believe that we can not solve the CRF problem by calling the constraint solver polynomial times. Therefore it is unlikely for us to have an efficient algorithm which can always generate all the logic relations for a constraint set.

3 The Rule Finding Algorithm

We can make use of linear programming solvers to decide if a clause is unsatisfiable. Due to the exponential size of the power set, we need to use some heuristics to speed up the whole process. The purpose of these heuristics is to reduce the number of times the linear programming solver is invoked.

3.1 The Basic Algorithm

First we give some notations. We use C to denote the set of numeric constraints and the size of C is denoted by NC. Let $C = \{c[1], \ldots, c[NC]\}$. We use two sets S and U to represent the sets of clauses that are satisfiable and unsatisfiable, respectively. S_i and U_i are the sets of satisfiable and unsatisfiable clauses found in loop i.

We use a dynamic programming method to enumerate all the formulae and construct the sets U and S. At first the set S has only one item $TRUE$ that can be regarded as a clause of length 0, and U is empty. For each element of C, add it (and its negation, respectively) to S and call `lp_solve` to solve the new set of constraints. Then we add the unsatisfiable formulae to U and the others to S. The algorithm is complete and will terminate after we have processed all the elements of C. The algorithm can be represented by a function in Fig 1.

```
void FindRule(){
    for($i = 1$;  $i \leq NC$;  $i$++){
        for each clause $s \in S$ {
            $w_1 = s \wedge c[i]$, $w_2 = s \wedge \neg c[i]$;
            if ($w_1$ is unsatisfiable) add $w_1$ to $U_i$ and $w_2$ to $S_i$;
            else {
                add $w_1$ to $S_i$;
                if ($w_2$ is satisfiable) add it to $S_i$;
                else add $w_2$ to $U_i$.
            }
        }
        $S = S \cup S_i$,  $U = U \cup U_i$;
    }
}
```

Fig. 1. The main algorithm

3.2 Correctness of the Algorithm

First we prove

Lemma 1. *The set S collects all the satisfiable clauses.*

Proof. Assume we have a satisfiable conjunctive clause

$$conj = C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_m} \text{ where } j_1 < j_2 < \ldots < j_m$$

and $conj \notin S$, we can easily know $C_{j_1}, C_{j_2}, \ldots C_{j_m}$ are all satisfiable.

Since $TRUE \in S$, then the formula $C_{j_1} = (TRUE \wedge C_{j_1}) \in S$. So $conj$ has at least one subclause in S.

Suppose we have two satisfiable subclauses of $conj$:

$$conj_k = C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_k} \text{ and } conj_{k+1} = C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_k} \wedge C_{j_{k+1}}$$

where $1 \leq k < m$, $conj_k \in S$ and $conj_{k+1} \notin S$. But according to our algorithm, the satisfiable $conj_{k+1}$ will be judged at the loop $i = j_{k+1}$ and must be added to S. So any satisfiable clause $conj$ will be added to set S. □

We say a rule is redundant if it has an unsatisfiable subclause. Now we prove our algorithm collects all the rules except some redundant ones.

Theorem 1. *Any unsatisfiable clauses will be included in the set U or it has an unsatisfiable subclause in the set U.*

Proof. For each unsatisfiable clause

$$conj = C_{j_1} \wedge C_{j_2} \ldots \wedge C_{j_m} \text{ where } j_1 < j_2 < \ldots < j_m,$$

if it has no unsatisfiable sub-formulae, its subclause $conj_{m-1}$ must belong to S and $conj$ should be added to U at the loop $i = j_m$. So the set U will collect all the unsatisfiable clauses except for some redundant ones. □

3.3 A Processing Example

We use Example 1 to show the search process. The clauses added to the two sets U or S at each step of the main loop are given in Table 1. We apply our search algorithm to Example 1 and quickly find that $C1 \wedge C2 \rightarrow C3$ is the only rule.

Table 1. The clauses added to U or S in each loop

The i'th loop	Clauses added to U	Clauses added to S
1	-	$C1, \neg C1$
2	-	$C2, \neg C2, C1 \wedge C2, C1 \wedge \neg C2, \neg C1 \wedge C2, \neg C1 \wedge \neg C2$
3	$C1 \wedge C2 \wedge \neg C3$	$C3, \neg C3, C1 \wedge C3, C1 \wedge \neg C3, \neg C1 \wedge C3, \neg C1 \wedge \neg C3,$ $C2 \wedge C3, C2 \wedge \neg C3, \neg C2 \wedge C3, \neg C2 \wedge \neg C3,$ $C1 \wedge C2 \wedge C3, C1 \wedge \neg C2 \wedge C3, C1 \wedge \neg C2 \wedge \neg C3$ $\neg C1 \wedge \neg C2 \wedge C3, \neg C1 \wedge \neg C2 \wedge \neg C3,$ $\neg C1 \wedge \neg C2 \wedge C3, \neg C1 \wedge \neg C2 \wedge \neg C3$

In the worst case, if the constraint set has no rules (or we can say this instance is very difficult), we have to check all the elements of the power set 3^C. That is to say, we have to solve 3^{NC} conjunctive clauses. Therefore the time complexity of this algorithm is exponential. To make the time cost of this algorithm acceptable, we developed some heuristics listed below to speed it up. The main idea of these techniques is to use other simple preprocessing methods instead of solving the clause to decide its satisfiability.

3.4 Subclause Strategy

Our algorithm has ruled out some redundant formulae which have subclauses in set U, but not all the redundant formulae. For example, if we find $C1 \wedge C3$ is unsatisfiable, then the formula $C1 \wedge C2 \wedge C3$ is obviously unsatisfiable. But it is still possible for the basic algorithm to check the formula. We should devise some technique to avoid checking all the redundant formulae during the search.

Our Subclause Strategy is based on the following observation: If A is a subclause of B and A is unsatisfiable, then B is unsatisfiable.

So our algorithm can be improved in the following way. We first sort the sets S and U by ascending order of clause length, then check if one of the clauses in U is a subclause of the current clause. For the clauses w_1 and w_2, since s has no subclauses in U, we only need to check whether w_1 and w_2 have subclauses in U_i.

3.5 Resolution Principle

Suppose we have two propositional logic expressions: $C_1 \vee p$ and $C_2 \vee \neg p$, where p is a propositional variable. Then we can obtain a new expression $C_1 \vee C_2$ that does not involve the variable p, while preserving satisfiability. That's the Resolution Principle of the propositional logic.

We will modify this deduction rule to form a pruning strategy.

Theorem 2. *Let A and B be two clauses. If we can express A and B as $A = A' \wedge p$* and *$B = B' \wedge \neg p$ where p is a logic variable, let $D = A' \wedge B'$. Assume A is unsatisfiable, then we have the following conclusions:*

I *B is unsatisfiable implies D is unsatisfiable;*
II *D is satisfiable implies B is satisfiable.*

This heursitic can be used to decide some clauses' satisfiablity. It can also be used to construct some satisfiable or unsatisfiable clauses. And if a rule is found, a series of rules will be soon generated and that will significantly prune the rest of the search space and speed up the whole process.

3.6 Related Numerical Constraints

The previous two strategies make use of the processed unsatisfiable formulae. In addition, some characteristics of arithmetic expressions can help us to check a clause's satisfiability quickly.

We say two clauses A and B are *related* if they have common numeric variables. For example, the three constraints of Example 1 are all related, while the following two constraints:

```
Cxy = (x == y);  Cz = (z > 0);
```

are not related. Their numeric variable sets $\{\texttt{x, y}\}$ and $\{\texttt{z}\}$ do not have any common element. The constraint $Cxy \wedge Cz$ is obviously satisfiable since Cxy and Cz are both satisfiable.

We summarize this simple principle here:

Theorem 3. *If a clause A can be divided into the conjunction of some satisfiable subclauses $A = A_1 \wedge \ldots \wedge A_k$, and any A_i and A_j $(i \neq j)$ are not related, then A is satisfiable.*

The premises of this strategy can be easily checked in our processing. Firstly, if the subclause stractegy is applied, then obviously all the subclauses are satisfiable. Secondly, a conjunction can be represented as a graph $\langle V, E \rangle$, where the V represents the set of numerical constraints. Two vertices have an edge if they are related. So the problem deciding if a conjunction can be divided is transformed to check the connectivity of an undirected graph, which can be done in polynomial time.

This heuristic is effective when dealing with the sets that each numeric constraints of them have few variables such that two numerical constraints have a considerable probability to be not related.

3.7 Linear Independency of Coefficient Vectors

In our search, we need to process some clauses with the same numeric constraints. For example, in Table 1, in the second loop, $C1 \wedge C2$, $C1 \wedge \neg C2$, $\neg C1 \wedge C2$, $\neg C1 \wedge \neg C2$ have the same constraints $C1$ and $C2$. These clauses have the same coefficient matrix A. For the clause $A\boldsymbol{x} \bowtie \boldsymbol{b}$, if there exists $\boldsymbol{b}' \bowtie \boldsymbol{b}$ such that $A\boldsymbol{x} = \boldsymbol{b}'$ is satisfiable, then the clause is also satisfiable. A set of vectors $\boldsymbol{a}_j$ $(1 \leq j \leq m)$ is *linear dependent* if there exist m factors λ_j $(1 \leq j \leq m)$, not all of which are zero, such that $\sum_{1\leq j\leq m} \lambda_j \boldsymbol{a_j} = 0$.

We have the following lemma:

Lemma 2. *For a clause $A\boldsymbol{x} \bowtie \boldsymbol{b}$, if the m coefficient vectors $\boldsymbol{a}_i$ $(1 \leq i \leq m)$ are linear independent, then the clause is satisfiable.*

This lemma only need to consider the coefficient matrix, so all the clauses involving the same constraints are satisfiable if the coefficient vectors are linear independent.

However, the constraints of linear dependency can not easily be solved since the solver `lp_solve` does not work well at the strict inequality $\lambda_j \neq 0$, so we strengthen our previous lemma to the following strategy:

Theorem 4. *For w_1 and w_2 of the main algorithm in Figure 1, if the coefficient vector $\boldsymbol{a}_c$ of $c[i]$ can not be linearly represented by the coefficient vectors of s, then w_1 and w_2 are satisfiable.*

Proof. Let $\boldsymbol{a}_j$ $(1 \leq j \leq m)$ denote the coefficient vectors of s, $A_i = [\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m]^{\mathrm{T}}$ be the coefficient matrix of s and $\boldsymbol{b}_s$ be the constant vector. If $\boldsymbol{a}_j$ $(1 \leq j \leq m)$ are linear independent, then $\boldsymbol{a}_c$ and $\boldsymbol{a}_j$ $(1 \leq j \leq m)$ are linear independent, thus w_1 and w_2 are satisfiable according to Lemma 2.

If $\boldsymbol{a}_j$ $(1 \leq j \leq m)$ are linear dependent, since $c[i]$ and s are satisfiable, so we have a constant vector $\boldsymbol{b}'_s \bowtie \boldsymbol{b}_s$, $r(A_i) = r([A_i, \boldsymbol{b}'_s])$. Here $r(A)$ means the *rank* of matrix A. Consider the constraints

$$\begin{bmatrix} \boldsymbol{a}_c \\ A_i \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} b'_c \\ \boldsymbol{b}'_s \end{bmatrix}$$

where b'_c is an arbitrary numeric value. Since $\boldsymbol{a}_c$ cannot be linearly represented by $\boldsymbol{a}_j$ $(1 \leq j \leq m)$, we have

$$r\left(\begin{bmatrix} \boldsymbol{a}_c \\ A_i \end{bmatrix}\right) = r(A_i) + 1 = r([A_i, \boldsymbol{b}'_s]) + 1 = r\left(\begin{bmatrix} \boldsymbol{a}_c & b'_c \\ A_i & \boldsymbol{b}'_s \end{bmatrix}\right)$$

that implies the constraint $\boldsymbol{a}_c\boldsymbol{x} = b'_c \wedge A_i\boldsymbol{x} = \boldsymbol{b}'_s$ is satisfiable and thus $w_1 = s \wedge c[i]$, $w_2 = s \wedge \neg c[i]$ are all satisfiable.

The strategy needs to call the solver once to judge the linear representation. This strategy aims at finding the satisfiable clauses and therefore it is efficient for the constraint sets for which only a few rules can be found. It will save at most $2^m - 1$ callings of the solver where m is the length of clauses. Please refer to Table 3 for the computational results of this strategy.

3.8 Clause Length Restriction

The heuristics introduced before are used for complete search. That is to say, they do not remove the useful rules. These techniques do not decrease the time complexity of the whole search. Here we introduce an approximate technique that can reduce the processing time efficiently.

Theoretically, the maximum length of rules is NC, which is the size of the numerical constraint set. In practice, we do not need to combine arbitrary number of numeric constraints. Firstly, most solvers are more efficient in dealing with short constraints. Secondly, we can see from the experiments 4.3 that the lengths of most clauses are in the domain $[1, NV + 1]$, where NV is the number of numerical variables. So we need not waste time in finding long formulas. We define UL as the upper bound on the formula's length. The clauses with length greater than UL will be abandoned. And the time complexity will be reduced to $\binom{NC}{UL} = O(NC^{UL})$. In practice, we can get a quicker process by choosing a small value for the parameter UL. For the empirical values of UL, please refer to Section 4.3.

4 Experimental Results

We developed a tool called MALL (MAchine Learning for Linear constraints) in the C programming language with all the presented techniques implemented. It invokes lp_solve to decide the satisfiability of clauses. Our tool can generate all the rules of a small constraint set without redundant rules quickly.

We have studied many examples including some random constraint sets. We mainly care about the number of solutions and the solving times. Some experimental results are listed here. We use a PC P4 3.2GHz CPU, 1 GB memory with Gentoo Linux, and the timings are measured in seconds.

4.1 A Real Instance

This strong correlative example comes from [5]. These constraints are collected from a real Middle routine to find the middle value of 3 values. For simplicity, we do not restrict the variables to be integers.

```
bool ba = (b < a);              bool ac = (a < c);
bool ca = (c < a);              bool ab = (a < b);
bool bc = (b < c);              bool cb = (c < b);
bool a_b = (a == b);            bool b_c = (b == c);
bool c_a = (c == a);            bool m_a = (m == a);
bool m_b = (m == b);            bool m_c = (m == c);
bool n_a = (n == a);            bool n_b = (n == b);
bool n_c = (n == c);            bool n_m = (n == m);
```

There are 16 constraints with 5 numerical variables. We tried our tool on this instance with different UL (upper bound of the formula length. See section 3.8). The results are summarized in Table 2. The result of $UL = 16$ reports all the useful logic relations among these constraints.

To check whether an implementation of Middle routine violates the specification, we can employ BoNuS to solve the following Boolean combinations of the numerical constraints:

Table 2. Different UL

UL	2	3	4	5	6	7	16
Number of solutions	9	92	260	557	869	887	887
Time used	0.01	0.11	0.43	2.11	11.55	37.95	184.71
Memory used(M)	1.11	1.56	3.60	12.47	38.11	91.36	444.23
Times to call `lp_solve`	135	548	892	1174	1244	1279	1282

```
% Specification
imp(and(ba, ac), m_a);  imp(and(ca, ab), m_a);
imp(and(ab, bc), m_b);  imp(and(cb, ba), m_b);
imp(and(ac, cb), m_c);  imp(and(bc, ca), m_c);
imp(b_c, m_a);  imp(c_a, m_b);  imp(a_b, m_c);

% Implementation
imp(or(and(ab, bc), and(cb, ba)), n_b);
imp(or(and(ac, cb), and(bc, ca)), n_c);
imp(and(not(or(and(ab, bc), and(cb, ba))), not(or(and(ac, cb),
and(bc, ca)))), n_a);

% Violation of the specification
not(n_m);
```

Here `imp(x,y)` denotes that `x` implies `y`, and a line starting with "`%`" is a comment. We apply `BoNuS` to this problem and `BoNuS` reports a solution in 0.084 second. If we add the rules with $UL = 3$ to the input of `BoNuS`, `BoNuS` will report a solution in 0.008 second. But if we add the rules with $UL = 5$ to `BoNuS` input, the solving time will be 0.012 second. In fact, for this type of mixed constraint satisfaction problem, some very short lemmas (rules) are enough. Too many long rules will decrease the efficiency because the solver has to spend more time on the logic constraints. If these 16 numerical constraints are checked repeatedly in program analysis, the 0.11 second of preprocessing is worthwhile.

To access the clauses (mainly the elements of S, U and some other clauses) quickly during the search, we use an index tree to record the processed clauses. This tree occupies too much memory if the clause is too long (please refer to Table 2 for memory used). That limits the scalability of the tool.

Our tool reports the rules quickly for some small UL value. But the space and time cost increases notably if UL is increased. On the other hand, the times to call `lp_solve` do not increase so remarkably as the memory cost increases. This is not possible if no heuristic is used.

Also we can see from this instance that $UL = 6$ is enough. In fact, we have experimented with some other small instances derived from real programs. The results are satisfactory. We can get more than 95% rules with proper UL in a few minutes if the number of constraints is not more than 16.

4.2 The Efficiency of Strategies

We also use the example of Section 4.1 with different UL to test the efficiency of four strategies: SC (Subclause Strategy), RP (Resolution Principle), RN (Related Numerical

Constraints) and LC (Linear Independency of Coefficient Vectors). Each time we remove a strategy and at last we remove all the strategies. Since we should check SC before RN, RN should be removed if SC is removed. We list the processing time and the number of rules found in Table 3. Note that some redundant rules are found if the SC strategy is removed.

Table 3. Efficiency of Strategies

Strategies Removed	None	SC+RN	RN	RP	LC	ALL
$UL = 3$	0.11 / 92	0.18 / 172	0.14 / 92	0.19 / 92	0.22 / 92	0.85 / 172
$UL = 4$	0.43 / 260	1.23 / 1571	0.49 / 260	1.35 / 260	0.61 / 260	5.97 / 1571
$UL = 5$	2.11 / 557	16.78 / 11107	2.16 / 557	8.91 / 557	2.20 / 557	34.48 / 11107

4.3 Effects of Various Parameters

In general, the smaller the rule set is, the more difficult the CRF problem is. Finding rules from a difficult set of constraints may waste much time and the rules may provide little useful information to the following constraint processing.

To study the characteristics of the CRF problem, we use some random instances. Here are some parameters to describe a set of linear constraints:

NC The number of numerical constraints.
NV The number of numerical variables.
ANV The ratio of average number of variables of each constraint to NV. It can be defined as $ANV = \frac{\sum NV_i}{NV*NC}$ where NV_i is the number of numerical variables of the i'th constraint.
NE The ratio of the number of equations and strict inequalities (whose relational operator is "$\neq$") to NC.

Obviously, with the increasing of NC, we will get more rules, but at the same time, the space and time cost will increase remarkably. In the following part we generate some random instances with different parameters to study how the latter 3 parameters influence the difficulty of the problem. Each random constraint is non-trivial (i.e. each numerical constraint and its negation are satisfiable). For each set of parameter values, the result is the average value of 100 runs.

Number of Numerical Variables. We use some 10-sized random sets (i.e., $NC = 10$) to study the influence of NV on the distribution of the conjunctive clause length. The results are given in Table 4.

From the results, we can conclude that, if $NV < NC$, most formulae have length $NV + 1$. Too small or too large NV will not cause many rules. We get the maximum number of rules near the point $NV = 4$. Our other experiments of small sized constraint set also indicate that we will get the maximum number of rules near the median of NC. These phenomena are mainly caused by two reasons:

Table 4. The Effect of the Number of Variables

Length of Clause	2	3	4	5	6	7	8	9	10	Total
$NV = 2$	10.75	53.46	0.00							64.21
$NV = 4$	1.86	3.76	8.36	117.79	0.00					131.77
$NV = 6$	0.62	0.70	1.15	1.07	4.29	70.5	0.00			78.33
$NV = 8$	0.26	0.20	0.18	0.29	0.49	0.29	0.62	6.49	0.00	8.82
$NV = 10$	0.12	0.07	0.06	0.11	0.06	0.11	0.06	0.00	0.00	0.59

1. The NV defines the dimensions of the variables' domain, which is a Euclidean space. According to linear programming, a set of standard inequalities have solution if we can get a solution along the edges of boundary. That is to say, the solvers just check the solutions of boundary constraints (which are a set of equations derived from the original inequalities just by replacing the relational operators with "="). According to linear algebra, $NV + 1$ linear equations of NV dimensions are linear dependent. That is why so many rules have length $NV + 1$.
2. Generally speaking, a small NV implies that these constraints are located in a "small" (low dimension) space and have more opportunities to be related. But on the other hand, because the number of numerical variables is small, many of these relations are redundant. Therefore, we may get a maximum number of rules in the middle of $[1, NC]$.

So from the results, we can get the experimental value of $UL = NV + 1$, and we have no need to find rules whose length is more than that.

Number of Equations. The constraints of equations usually come from the assignments of program. Here we use some instances with 10 constraints to test its effects on the difficulty of the problem. In these tests, the UL value is set as $NV + 2$ to get the most rules, and the results are listed in Table 5.

Table 5. The Effect of Equations

NE	0.2	0.4	0.6	0.8
Number of Rules	58.83	58.23	55.97	58.26
Times calling `lp_solve`	4303.88	4471.27	4911.94	5573.89
Time used	1.26	1.33	1.45	1.65

From the result, we see that the parameter NE has no significant effect on the problem's difficulty. In practice, a strict inequality is translated into two inequalities, and it will cost more time.

Average Number of Numerical Variables. The parameter ANV affects the size of rules. Here we also use some random constraint set sized 10 to test it. The value NE is set as 0.33. The results are summarized in Table 6.

Table 6. The Effect of ANV

ANV	0.2	0.4	0.6	0.8
Number of rules	14.73	44.12	80.09	98.31
Times calling `lp_solve`	698.20	4453.92	6200.64	6481.27
Time used	0.22	1.25	1.83	1.99

From the result, we get that the number of rules increases as ANV increases. Meanwhile, the time cost is also increased. The main reason is that the constraints with high ANV tend to be related.

5 Related Work

There are many research work about analysis of numerical constraints. Constraint Logic Programming systems (e.g. CLP($\mathcal{R}$)[12]) can find solutions if constraints are satisfiable or detect unsatisfiability of constraints. This type of systems focus on checking the satisfiability of mixed types of constraints. Different from our constraint solving method, these works treat equations and inequalities differently. Also on the analysis of linear constraints from programs, the paper [13] presents a method for generating linear invariants for transition systems. Compared with our work, this type of works focus on inducing invariants (which are some numerical expressions) from all the constraints, while we try to find all the logic rules that each may fit for only a small subset of the constraints. Our work, if properly modified, can be used as preprocessing in these systems.

The Inductive Logic Programming (ILP) [14] is a research area formed at the intersection of Machine Learning and Logic Programming. ILP systems have been applied to various learning problem domains. ILP systems develop predicate descriptions from examples and background knowledge. The examples, background knowledge and final descriptions are all described as logic programs. The theory of ILP is based on proof theory and model theory for the first order predicate calculus. In many occasions, the ILP background knowledge can be described as a series of numerical constraints. If we can find the relations between these constraints, we can remove much redundancy and reduce the time of induction or searching.

Conflict-driven Lemma Learning [15] has been proved quite successful in improving SAT algorithms. This type of techniques recognize and record the causes of conflicts, preempt the occurrence of similar conflicts later on the search. When solving complex constraints, recognizing the cause of conflicts is difficult due to the numeric constraints. So we use a pre-learning approach as described in this article instead of dynamic analysis. Also in solving a constraint problem, sometimes we need to add redundant constraints which can lead to new simplifications. For example, the paper [2] examined the impact of redundant domain constraints on the effectiveness of a real-time scheduling algorithm. But the forms of redundant constraints are restricted.

Similar to our CRF problem, the problem MUS (finding minimal unsatisfiable subformula) is used in the solution of SAT and other problems. Some approximate search algorithms for MUS have been developed. For example, the paper [16] discussed an adaptive search method to find the MUS of an unsatisfiable CNF instance.

6 Conclusion

In this paper we study the problem of finding rules from a set of linear constraints and give its theoretic complexity. We present a search method to solve this problem efficiently and developed a tool to generate the rules automatically. Since we can not get all the rules in polynomial-time, we propose a number of strategies to speed it up. Most of these strategies are independent of the constraint type, therefore, our method is general and can be used on other kind of constraints if there are proper solvers.

We also study the difficulty of the problem using some randomly generated instances. We find that some parameters of the instances affect the number of rules notably. We obtain some empirical values to increase the performance of our tool.

Future works are needed to improve the proposed method. Firstly, the algorithm introduced in this paper still has some trouble in processing large scale constraint sets. This deficiency may be resolved by some approximate techniques such as dividing the original constraint set into several small subsets:

$$S = S_1 \cup S_2 \ldots \cup S_k$$

and processing each subset separately. Secondly, we can try other type of constraints. Each constraint can be non-linear, a Boolean combination of numerical constraints or some other kind of mixed constraint. At last, we still need some efficient heuristics to improve the processing speed.

Acknowledgements

The authors are very grateful to the anonymous reviewers for their detailed comments and suggestions.

References

1. Cheng, B.M.W., Lee, J.H.M., Wu, J.C.K.: Speeding up constraint propagation by redundant modeling. In: Proceedings of CP-96, LNCS 1118. (1996) 91–103
2. Getoor, L., Ottosson, G., Fromherz, M., Carlson, B.: Effective redundant constraints for online scheduling. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence(AAAI'97). (1997) 302–307
3. Zhang, J., Wang, X.: A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering & Knowledge Engineering **11** (2001) 139–156
4. Zhang, J.: A path-based approach to the detection of infinite looping. In: Second Asia-Pacific Conference on Quality Software (APAQS'01). (2001) 88–94
5. Zhang, J.: Specification analysis and test data generation by solving Boolean combinations of numeric constraints. In: Proceedings of the First Asia-Pacific Conference on Quality Software, Hong Kong. (2000) 267–274
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems **16** (1994) 1512–1542
7. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: Proc. of the 7th International SPIN Workshop, LNCS 1885. (2000) 113–130

8. Berkelaar, M.: `LP_solve` (May 2003) A public domain Mixed Integer Linear Program solver, availiable at `http://groups.yahoo.com/group/lp_solve/`.
9. Karmarkar, N.: A new polynomial-time algorithm for linear programming. Combinatorica **4** (1984) 373–395
10. Papadimitriou, C.H., Wolfe, D.: The complexity of facets resolved. Journal of Computer and System Sciences **37** (1988) 2–13
11. Papadimitriou, C.H., Yannakakis, M.: The complexity of facets (and some facets of complexity). In: Proceedings of the fourteenth annual ACM symposium on Theory of computing (STOC '82). (1982) 255–260
12. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C.: The CLP($\mathcal{R}$) language and system. ACM Transactions on Programming Languages and Systems (July 1992) 339–395
13. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Proceedings of VMCAI 2005, LNCS 3385. (2005)
14. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19/20** (1994) 629–679
15. Silva, J., Sakallah, K.: Conflict analysis in search algorithms for propositional satisfiability. In: Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI '96). (1996) 467
16. Bruni, R., Sassano, A.: Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In: Proceedings of the Workshop on Theory and Applications of Satisfiability Testing. (2001)