HAL
open science

A general algorithm for data dependence analysis

Christine Eisenbeis, Jean-Claude Sogno

To cite this version:

Christine Eisenbeis, Jean-Claude Sogno. A general algorithm for data dependence analysis. [Research Report] RR-1699, INRIA. 1992. inria-00076936

HAL Id: inria-00076936

https://inria.hal.science/inria-00076936v1

Submitted on 29 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

N°1699

Programme 2

Calcul symbolique, Programmation et Génie logiciel

A GENERAL ALGORITHM FOR DATA DEPENDENCE ANALYSIS

Christine EISENBEIS
Jean-Claude SOGNO

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Mai 1992

A GENERAL ALGORITHM FOR DATA DEPENDENCE ANALYSIS *†

UN ALGORITHME GENÉRAL POUR L'ANALYSE DES DÉPENDANCES DE DONNÉES

Christine Eisenbeis ‡
Jean-Claude Sogno §

INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex
FRANCE

*This work was partially supported by ESPRIT Project COMPARE
†Contributed Paper to "International Conference on Supercomputing 1992", Washington, July 19-23, 1992
‡e-mail:Christine.Eisenbeis@inria.fr
§e-mail:Jean-Claude.Sogno@inria.fr

Abstract

With the development of ever more sophisticated data flow analysis algorithms, traditional data dependence tests based on elementary loop information will not be sufficient in the future. In this paper, quite general algorithms are presented for solving integer linear programming problems. While the properly so called problem solution is performed by a standard algorithm (the dual all integer algorithm), preliminary problem reduction techniques not only serve as a powerful tool for preparing this latter step, but also are often sufficient for solving exactly the data dependence problem.

Résumé

Avec le développement d'algorithmes d'analyse de flot de données de plus en plus sophistiqués, il apparaît que les tests traditionnels d'analyse de dépendances, basés sur des informations élémentaires sur la boucle, ne seront plus suffisants à l'avenir. Dans ce papier, nous présentons des algorithmes généraux de résolution de programmes linéaires en nombres entiers. La résolution proprement dite du programme linéaire est effectuée par un algorithme standard ("dual tout entier"). Les techniques de réduction du probleme utilisées dans une phase préliminaire se revèlent un outil puissant, non seulement pour faciliter cette dernière étape, mais souvent aussi tout simplement pour résoudre le problème de dépendance de données de manière exacte.

1 Introduction

The power of a parallelizer largely depends on its capacity to disambiguate precisely memory references and therefore to decide whether two portions of code may be run in parallel or not. In the case of loop data dependence analysis, one is given a nested loop with indices i_1, i_2, ..., i_k (Figure 1). This loop may or may not be perfectly nested. The question is whether two statements are likely to access the same memory location, possibly enforcing an execution order on the two statements. Let the statements S_1 and S_2 access array X array via the linear functions $h(i_1, i_2, \cdots, i_k)$ and $g(i_1, i_2, \cdots, i_k)$ respectively. Then there is a dependence between S_1 and S_2 if and only if there exist integers i_1, i_2, ..., i_k and j_1, j_2, ..., j_k such that:

Loop Bounds

$$i_p, j_p \in [L_p, U_p] \tag{1}$$

(Note that L_p and U_p may depend on previous indices i_1, i_2, ..., i_{p-1}.)

Memory Collision

$$h(i_1, i_2, \cdots, i_k) = g(j_1, j_2, \cdots, j_k) \tag{2}$$

Now that the basic problem is stated, let us explain which properties are expected from a data dependence analyzer.

The first property which has always been emphasized until now, was the rapidity of the test. That is why the usual methods profit by the specific statement of the problem, as given

above. They are mainly based on relaxation of constraints. The GCD test [Ban79] does not consider loop bounds, while the Banerjee test [Ban79] disregards integrity constraints on indices.

The second property is the degree of test accuracy desired. Accuracy is related to three points. First, it depends on the answer we want. [Wol82], for instance, determines loop direction vectors, i.e. he searches for dependence such that $(i_1, i_2, \cdots, i_k)$ is less than $(j_1, j_2, \cdots, j_k)$ in the lexicographic order ("$<$" dependence). Therefore new equations or inequalities have to be added to the original system. Second, it depends on the precision of the algorithm performed. Indeed, three answers are possible:

1. No, there is no solution. The system is incompatible and it is proven that there is no dependence.

2. Yes, there is a solution. The statements are dependent.

3. Maybe, we have not been able to prove or disprove the existence of a solution.

In the first two cases, the right solution is produced. The result is *exact*. In the latter one, it is inexact. In this case, we must assume a dependence exists in order to preserve the program semantics. This happens because the general integer linear programming problem is NP-complete. It may therefore be computationally impossible or too expensive to decide the existence of a solution or not. Heuristics must be used.

Third, accuracy depends on the knowledge we have about data semantics. Actually, more and more sophisticated techniques are emerging in the domain of data flow analysis. Besides the usual constant propagation [WZ85], [CC77] was able to determine intervals for variable variations. Then [CH78] tackled the problem of determining linear equalities relating variables. More recently, [Gra89] designed an algorithm for discovering "modulo" linear equalities between variables. Such precisions about variable range must be incorporated into the system, since they are likely to make the system incompatible and therefore to ensure independence between statements. Accuracy depends also on the quality of approximation performed to linearize nonlinear systems (intrinsically nonlinear, or when coefficients values are not known at compile time). The algorithm we present in this paper is coupled to symbolic problem solution described in [LT88].

Unlike previous work, we have chosen to stress the problem of accuracy. That is why instead of considering linear integer systems like that in figure 2, we come back to the general integer programming problem and propose a general algorithm for solving such a problem, actually the dual all integer algorithm. However, we do not forget about the test rapidity requirement. The system is preprocessed through a succession of reductions steps speeding up the algorithm (this is a usual technique in mathematical programming). This preprocessing phase appears as a powerful tool, since it is sometimes sufficient to solve the system. Otherwise, the system is reduced and requires less iterations of the dual all integer algorithm to be solved.

The paper is organized as follows. In the first section, we recall previous work done in the domain of data dependence analysis. As a matter of fact, some traditional tests serve in the

preprocessing step of our algorithm. In the second section, we explain the other reduction steps used. The main requirement is that they all reduce to equivalent systems for the method to be exact. In the third section, some general methods for mathematical programming are presented. Some of them (Constrained matrix test and FAS3T) have already been applied to data dependency tests, while this is the first time, to our knowledge, that the dual all integer algorithm in used for such a purpose. The last section is devoted to the construction of a general algorithm for data dependence analysis based on the basic building blocks presented in the previous sections.

```
      DO 1 i_1 = L_1, U_1
            DO 1 i_2 = L_2, U_2
                  ...
                  DO 1 i_k = L_k, U_k
      < S_1 >     X[h(i_1, i_2, ..., i_k)]    =    ...
      < S_2 >     ...    =    X[g(i_1, i_2, ..., i_k)]
1     CONTINUE
```

Figure 1: Example of a k-nested loop

$$
\begin{array}{rcccl}
L_p & \leq & i_p & \leq & U_p \qquad p = 1, \ldots, k \\
L_p & \leq & j_p & \leq & U_p \qquad p = 1, \ldots, k \\
h(i_1, i_2, \cdots, i_k) & & = & & g(j_1, j_2, \cdots, j_k)
\end{array}
$$

Figure 2: Linear integer programming resulting from previous loop

2 Related Work

Many authors have studied data dependence test problem. In this section, we will class them into three categories, namely approximate tests, exact but data dependence oriented tests and, finally, general integer linear programming based tests. Some of the methods are detailed, since they are used later in the paper. Some other well-known methods are not mentioned; for a complete bibliography, the reader can refer to [GKT91].

Before saying anything about the methods, let us say something about the complexity of the problem. Contrary to the generally accepted opinion, integer linear programming satisfiability problem is not "so difficult". As a matter of fact, and this has been recalled by [MHL91], Lenstra [Len83] showed in 1983 that when considering a fixed number of variables, this problem is polynomial. The difficulty lies in the fact that complexity grows exponentially with the number of variables. In a linear programming problem which stems from a dependence test, the number of variables is about twice the depth of nested loop. By duality, this result is also true for a fixed number of equations or inequalities (therefore fixed dimensional arrays). Since these values are not too large in real programs, we think that this is a good argument for turning towards exact methods.

2.1 Approximate methods

In this section we assume that the system is composed of inequalities of type (1) or equations of type (2), unless explicitly mentioned.

2.1.1 GCD-Test (Loop bounds constraint relaxation)

This method [Ban79] consists in trying to disprove the feasibility of linear *equations* and is based on the following observation: if the equation $a_1 i_1 + a_2 i_2 + \ldots + a_k i_k = b$, where the a_i are integers, has a solution, then b is a multiple of the $\gcd(a_1, a_2, \ldots, a_k)$ [1]. The test consists in verifying that property. If it does not hold, the whole system is independent.

2.1.2 Banerjee Extended GCD-Test (Loop bounds constraints relaxation)

This method [Ban88] is the extension of previous one to multi-dimensional systems. It is based on the fact that an integer $m \times n$ matrix A can be factored as $A = DU$, where U is an integer unimodular matrix and D is a $m \times n$ echelon matrix (known as the reduced Hermite form of a matrix [Min83, Sch86]). Then the system $\vec{x}A = \vec{c}$ has an integer solution if and only if there exists an integer vector $\vec{t}$ such that $\vec{t}D = \vec{c}$. This can be determined very simply. In addition, [MHL91] remark that working with $\vec{t}$ instead of $\vec{x}$ permits the reduction of the number of variables and constraints (hence the term "parameterization"). It should be pointed out that some steps of section 3 can be considered as a kind of incomplete parameterization. We prefer to split the whole matrix decomposition algorithm into such steps because it often permits the elimination, thereby to simplifying the system and sometimes even leading to immediate termination.

2.1.3 Banerjee Test(Integrity relaxation)

This method [Ban79] consists of trying to disprove the feasibility of equations, based on the loop bounds. For simplicity, let us assume that the coefficients of the equations a_i are

[1] "gcd" or "GCD" stands for the "greatest common divisor"

non-negative. From the inequalities:

$$L_j \leq i_j \leq U_i, i = 1, \ldots, k$$

it follows that, if there is a solution to:

$$a_1 i_1 + a_2 i_2 + \ldots + a_k i_k = b$$

then b must be bounded by $a_1 L_1 + a_2 L_2 + \ldots + a_k L_k$ and $a_1 U_1 + a_2 U_2 + \ldots + a_k U_k$. If this is not true, then the system is independent. Otherwise, we can conclude that there exists a *real* solution, but not necessarily an *integer* one. The system is assumed dependent.

We will not use this technique in our work. However, it should be mentioned that in [KPK90], a sufficient and necessary condition for Banerjee test to be exact, which is based on coefficients and loop bounds, is given. This in turn has led to a more accurate algorithm based on the iterative application of Banerjee and GCD tests, the I-test [KKP91]. This method is still inexact in general.

2.2 Exact data dependence oriented tests

Only recently authors have tried to develop exact methods for data dependence analysis. Main contributions can be attributed to [MHL91, GKT91]. They both are based on the successive application of exact and computationally inexpensive tests. When tests do not suffice to obtain a solution then [MHL91] use a (not polynomial) Fourier-Motzkin algorithm and [GKT91] a (not exact) so-called constraint propagation algorithm, whose exactness can be obtained at a not polynomial price. Based on analysis of well-know scientific benchmarks, both teams conclude that a first pass of quick tests often suffices to determine exactly the dependence or independence of the system.

2.3 Exact general satisfiability integer linear programming problem

While it seems quite natural to use classic techniques from the mathematical programming domain to solve the data dependence problem, only a few authors have tackled that problem from this angle. These methods are largely explained in section 4. The very basic underlying algorithm of all these methods is the well-known (real) simplex algorithm. There are many ways to adapt it to the integer domain. We think that the description of section 4 is quite instructive. The first method consists of applying as many simplex algorithms as the number of equations, by using the economic function (to be minimized) to force the equations to be true. To obtain integer results, cutting inequalities must be added to the system between two successive simplex applications. The second method applies to a system of inequalities and tries to enter the integral simplex domain by solving at each step one integer simplex with minimization function equal to the constraints that have not yet been verified. Our method uses the dual of integer simplex algorithm. We have chosen to use it because our experiments indicate that its performance on data dependence systems is good.

3 System Reduction

In this section some techniques for reducing the original problem are described. These reductions may sometimes appear as evident or trivial, or they may use already well-known data dependence analysis techniques. However in some cases, a simple algorithm consisting of a combination of the steps described here will be sufficient to answer the question of whether or not a integer solution to the original problem exists. This will be explained in section 5.

In the other cases, the transformed system is greatly simplified. The reduction phase is therefore a powerful tool for preparing the system for resolution by a general method described in section 4.

3.1 Integer Gaussian elimination

This technique eliminates one equation and one variable: if one of the coefficients of an equation has a unitary value (i.e. ± 1), a Gaussian elimination of the variable associated with that coefficient will build an equivalent and smaller integer system. The expression of the eliminated variable includes only integer coefficients, so we are guaranteed that the transformed system remains equivalent to the previous one. Let us consider the following system:

$$\begin{array}{rrrcr} x_1 & +x_2 & & \leq & 10 \\ & x_2 & -4x_3 & \leq & 12 \\ -6x_1 & +x_2 & -7x_3 & = & -4 \\ -4x_1 & +3x_2 & -2x_3 & = & -1 \end{array}$$

The first equation gives an integer expression of x_2:

$$x_2 = 6x_1 + 7x_3 - 4$$

Therefore, by replacing x_2, we derive the equivalent system:

$$\begin{array}{rrcr} 7x_1 & +7x_3 & \leq & 14 \\ 6x_1 & +3x_3 & \leq & 16 \\ 14x_1 & +19x_3 & = & 11 \end{array}$$

and we can now forget about x_2.

Since this can be applied when one coefficient is ± 1, many of others techniques described in this section are aimed at generating unitary coefficients.

3.2 Division by GCD

When the GCD test (2.1.1) fails and the GCD is not 1, then performing the (integer) division of all the coefficients by the GCD diminishes the value of coefficients. This will make the

other stages less complex. Furthermore, this is likely to make a ± 1 coefficient appear. For instance, the following equation:

$$4x_1 - 2x_2 + 4x_3 = 110$$

becomes

$$2x_1 - 1x_2 + 2x_3 = 55$$

after division by 2. Now the elimination of x_2 can be performed.

The same technique can be applied to inequalities. It makes following reduction steps described below (3.7 and 3.8) and the integer simplex methods easier, since coefficients of value ± 1 are desirable. For instance:

$$-8x_1 - 4x_2 + 2x_3 \leq -17$$

The inequality can be replaced by:

$$-4x_1 - 2x_2 + x_3 \leq -9$$

3.3 Mixing two equations

Systems can also be transformed by combining equations: it is straightforward to show that the two following equations:

$$\begin{array}{rcl} \sum_{k=1}^{n} a_{ik}x_k & = & d_i \\ \sum_{k=1}^{n} a_{jk}x_k & = & d_j \end{array}$$

can be replaced by the equivalent system:

$$\begin{array}{rcl} \sum_{k=1}^{n} a_{ik}x_k & = & d_i \\ \sum_{k=1}^{n} (\lambda_i a_{ik} + \lambda_j a_{jk})x_k & = & \lambda_i d_i + \lambda_j d_j \end{array}$$

provided that λ_j is non-zero. If we find λ_i and λ_j such that

$$(\lambda_i a_{ik} + \lambda_j a_{jk}) = \pm 1 \tag{3}$$

for one k, then x_k elimination can again be performed.

This can be achieved as soon as a_{ik} and a_{jk} are relatively prime, in which case the extended Euclid algorithm [Knu81] finds two such integers λ_i and λ_j.

Consider for instance the system:

$$\begin{array}{rrrcr} x_1 & +x_2 & & \leq & 10 \\ & x_2 & -4x_3 & \leq & 12 \\ -2x_1 & +5x_2 & +3x_3 & = & 2 \\ -4x_1 & +3x_2 & -2x_3 & = & -1 \end{array}$$

The coefficients of x_1's are -2 and -4. Since their GCD is 2, this technique does not work. However, the coefficients of x_2 are 5 and 3. Their GCD is 1 and we find that:

$$(-1)5 + (2)3 = 1$$

Thus, by multiplying the first equation by $\lambda_1 = -1$ and second one by $\lambda_2 = 2$ and summing them, we obtain:

$$\begin{array}{rrrcr} -6x_1 & +x_2 & -7x_3 & = & -4 \end{array}$$

Hence the new system is:

$$\begin{array}{rrrcr} x_1 & +x_2 & & \leq & 10 \\ & x_2 & -4x_3 & \leq & 12 \\ -6x_1 & +x_2 & -7x_3 & = & -4 \\ -4x_1 & +3x_2 & -2x_3 & = & -1 \end{array}$$

where now x_2 elimination can be performed.

3.4 Changing two variables

Instead of combining equations, it is also possible to "combine" two variables in some sense, by a judicious change of variables. Imagine that in the following equation

$$\sum_{j=1}^{n} a_j x_j \quad = \quad d \tag{4}$$

a_1 and a_2 are relatively prime. Then any integer t_1 can be written under the form $t_1 = a_1x_1 + a_2x_2$, where x_1 and x_2 are integers. This suggests the use of t_1 as a new variable. In order to maintain an equivalent system, a new variable t_2 must also be introduced. Variable changing is thus described by a transformation such as:

$$\left\{ \begin{array}{rcll} t_1 & = & a_1x_1 & +a_2x_2 \\ t_2 & = & b_1x_1 & +b_2x_2 \end{array} \right.$$

For this transformation to be unimodular, we must have

$$a_1b_2 - a_2b_1 = \pm 1$$

Again the extended Euclid algorithm provides two such integer coefficients b_1 and b_2 (if $\lambda_1a_1 + \lambda_2a_2 = 1$, we take $b_1 = -\lambda_2$ and $b_2 = \lambda_1$).

Now it is easy to find the inverse transformation:

$$\left\{ \begin{array}{rcll} x_1 & = & \lambda_1t_1 & -a_2t_2 \\ x_2 & = & \lambda_2t_1 & +a_1t_2 \end{array} \right.$$

Therefore, x_1 and x_2 are replaced by t_1 and t_2 everywhere in the system and equation (4) becomes

$$t_1 + \sum_{j=3}^{n} a_j x_j \quad = \quad d$$

where t_1 elimination can be performed.

Let us consider the previous example:

$$\begin{array}{rrrcr} x_1 & +x_2 & & \leq & 10 \\ & x_2 & -4x_3 & \leq & 12 \\ -2x_1 & +5x_2 & +3x_3 & = & 2 \\ -4x_1 & +3x_2 & -2x_3 & = & -1 \end{array}$$

In the last equation, we observe that we can apply this transformation to (x_1, x_2) or (x_2, x_3) (but not to (x_1, x_3) since the value of $GCD(-4, -2)$ is 2). We choose (x_1, x_2) for instance and obtain:

$$\begin{array}{rcll} x_1 & = & -t_1 & -3t_2 \\ x_2 & = & -t_1 & -4t_2 \end{array}$$

The system becomes:

$$\begin{array}{rrrcr} -2t_1 & -7t_2 & & \leq & 10 \\ -t_1 & -4t_2 & -4x_3 & \leq & 12 \\ -3t_1 & -14t_2 & +3x_3 & = & 2 \\ t_1 & & -2x_3 & = & -1 \end{array}$$

A Gaussian elimination is now possible with variable t_1.

3.5 Shortening the number of variables of an equation

Even when there is no pair of relatively prime coefficients, the technique of changing variables can be used to eliminate a variable. By applying this technique iteratively, we obtain the case when two coefficients are relatively prime, provided that the GCD of the coefficients of the original equation was 1. If $GCD(a_1, a_2) = p$, then we can create the new variable t_1 such that $pt_1 = a_1x_1 + a_2x_2$ i.e. $t_1 = \frac{a_1}{p}x_1 + \frac{a_2}{p}x_2$, which yields to the previous case. Then we compute λ_1 and λ_2 such that $\lambda_1\frac{a_1}{p} + \lambda_2\frac{a_2}{p} = 1$ and perform the following variable change:

$$\left\{ \begin{array}{rcll} t_1 & = & \frac{a_1}{p}x_1 & +\frac{a_2}{p}x_2 \\ t_2 & = & -\lambda_2 x_1 & +\lambda_1 x_2 \end{array} \right.$$

with inverse:

$$\left\{ \begin{array}{rcll} x_1 & = & \lambda_1 t_1 & -\frac{a_2}{p}t_2 \\ x_2 & = & \lambda_2 t_1 & +\frac{a_1}{p}t_2 \end{array} \right.$$

Now equation (4) becomes:

$$pt_1 + \sum_{j=3}^{n} a_j x_j \quad = \quad d$$

As an example, consider the following system:

$$\begin{array}{rrrcr} x_1 & +x_2 & & \leq & 10 \\ & x_2 & -4x_3 & \leq & 12 \\ -6x_1 & -10x_2 & +15x_3 & = & 2 \end{array}$$

In this example, there is no equation which contains a pair of coefficients which are relatively prime, so we consider (x_1, x_2) and compute the formula:

$$\begin{array}{rcrr} x_1 & = & -2t_1 & +5t_2 \\ x_2 & = & t_1 & -3t_2 \end{array}$$

The system becomes:

$$\begin{array}{rrrcr} -t_1 & +2t_2 & & \leq & 10 \\ t_1 & -3t_2 & -4x_3 & \leq & 12 \\ 2t_1 & & +15x_3 & = & 2 \end{array}$$

Now the two coefficients of the equation are relatively prime and the system can be reduced with the method given in section 3.4.

3.6 Fast GCD

The methods discussed above make intensive use of GCD calculations. Therefore, a fast algorithm for determining the GCD of two numbers is desirable. The principle of our method is to have in memory a table of precomputed GCD's, say a table of GCD of every pair of integers less than a given value m. Then, if it happens that the numbers whose GCD we want to compute are less that m, it is straightforward and does not cost anything to consult the table. If not, then the table can still serve for speeding up the GCD computation. To explain this, let us recall the Euclid algorithm for computing the GCD of two positive numbers a and b such that a is greater than b:

```
gcd (a,b) =
            { if  b = 0 then a
                        else gcd (b, a mod b) }
```

Now, if we have a table *tgcd* such that $gcd(a,b) = tgcd(a,b)$ for any a and b less than m, then the algorithm becomes:

```
gcd (a,b) =
            {if (b = 0) then a
                        else
                        { if  a ≤ m then tgcd (a,b)
                        else gcd (b, a mod b) } }
```

The extended Euclid algorithm for determining coefficients of equation 3 can be adapted in the same manner and profit by a table of precomputed Bezout's coefficients.

In systems issued from data dependence analysis, coefficients are often small. In that case, one memory reference suffices to compute GCD of two numbers. As a result, the methods described above run fast.

3.7 Selective Fourier-Motzkin

The general Fourier-Motzkin elimination method applies to a system of inequalities. Its principle is to eliminate every variable in turn, until none remain or until a contradiction is found. It is known to be rather computationally expensive, especially for large systems because it may generate an exponential number of additional inequalities. Furthermore, the result it gives is exact for real variables but not exact in general when variables are constrained to be integers.

However, in this section, we show that Fourier-Motzkin method can serve as a preprocessing step for reducing the system by partial elimination of variables, while still working in equivalent systems, hence keeping the method exact.

Let us see how the general method works: each variable is eliminated in turn by the following algorithm (suppose we want to eliminate x_1).

- In each inequality, x_1 is isolated, so that a lower or upper bound for x_1, depending on other variables, is deduced.
- The system is compatible only if all lower bounds for x_1 are less than all upper bounds for x_1.
- We are left with inequalities without the variable x_1 and we do the same for remaining variables $x_2, \ldots, x_k$.
- If we arrive at a contradiction, then the system is independent. Otherwise, there exists a (not necessarily integer) solution.

Let us split the m inequalities of the system into three classes, depending on the sign of x_1 coefficient: m_1 (resp. m_2) inequalities are such that x_1 coefficient is positive (resp. negative). The third consists of the inequalities in which x_1 does not appear. Then m_1 upper bounds and m_2 lower bounds for x_1 are derived. This gives rise to a new system of $m_1 m_2$ inequalities (coefficients remain integer by reducing both sides of the inequalities to the same denominator), hence the growing complexity of the system.

The uncertainty of the method lies in the fact that one interval may be non-empty and yet contain no integer. However, it is easy to prove [Sog92] that:

Theorem 3.1 *The transformed system is equivalent to the previous one when one of the following criteria is verified:*

Criterion 1 *The value of every negative coefficient of x_1 is -1*

Criterion 2 *The value of every positive coefficient of x_1 is 1*

Therefore, we will apply this reduction only if one of these criteria is verified. Still remains the problem of the number of inequalities generated. It is clear that it will not increase in any of these cases:

Case 1 $m_1 \leq 1$

Case 2 $m_2 \leq 1$

Case 3 $m_1 = 2$ and $m_2 = 2$

Therefore it is worth applying this method only in one of these three cases.

As an example, consider the following system:

$$\begin{array}{rrrcr} 2x_1 & -3x_2 & & \leq & 2 \\ -x_1 & +4x_2 & -2x_3 & \leq & -6 \\ & x_2 & +2x_3 & \leq & 8 \\ & & -x_3 & \leq & -5 \end{array}$$

The variable x_1 can be eliminated (Criterion 1, Case 1) and the two first inequalities (the only ones containing x_1) are replaced by one new inequality:

$$\begin{array}{rrcr} 5x_2 & -4x_3 & \leq & -10 \\ x_2 & +2x_3 & \leq & 8 \\ & -x_3 & \leq & -5 \end{array}$$

We can continue by eliminating the variable x_2 (Criterion 1, Case 1). The two first inequalities are removed and a single inequality remains:

$$-x_3 \leq -5$$

This last inequality has an obvious integer solution, therefore the initial system is compatible.

3.8 Introducing non-negative variables

Most algorithms used to solve the general linear integer programming problem (section 4) require that variables be "constrained", i.e. of non-negative value. Unconstrained variables will be designated as "free". Generally the problems as originally posed include free variables. Replacing directly free variables by constrained variables is always possible, but this classic technique increases the number of variables. Fortunately, inequalities involve generally unitary coefficients. Then, by introducing a slack variable ≥ 0, every such inequality can be replaced by an equality, yielding a possibility of Gaussian elimination.

Let us consider this system with free variables:

$$\begin{array}{rrrcr} -4x_1 & -2x_2 & +x_3 & \leq & -9 \\ -3x_1 & +x_2 & & \leq & -4 \\ x_1 & -x_2 & +2x_3 & \leq & -3 \\ 2x_1 & +2x_2 & -5x_3 & \leq & 5 \end{array}$$

Introducing a slack (dummy) variable $y_4 \geq 0$, the first constraint can be written:

$$y_4 - 4x_1 - 2x_2 + x_3 = -9$$

From this equation, we obtain an integer expression for x_3, and the new equivalent system:

$$\begin{array}{rrrcr} -3x_1 & +x_2 & & \leq & -4 \\ 9x_1 & +3x_2 & -2y_4 & \leq & 15 \\ -18x_1 & -8x_2 & +5y_4 & \leq & -40 \end{array}$$

A similar operation is possible with the upper inequality and the variable x_2:

$$y_5 - 3x_1 + x_2 = -4$$

The system is now:

$$\begin{array}{rrrcr} 18x_1 & -3y_5 & -2y_4 & \leq & 27 \\ -42x_1 & +8y_5 & +5y_4 & \leq & -72 \end{array}$$

Such an operation is not possible with the variable x_1. So we define two constrained variables:

$$x_1 = y_6 - y_7$$

and the final system of constrained variables is:

$$\begin{array}{rrrrcr} 18y_6 & -18y_7 & -3y_5 & -2y_4 & \leq & 27 \\ -42y_6 & +42y_7 & +8y_5 & +5y_4 & \leq & -72 \end{array}$$

More generally, if more than one free variable remains, it is possible to transform a system with n free variables x_i into a system with $n+1$ constrained variables by setting (classical "single added variable" technique):

$$x_i = y_i - y_0$$

4 Solving the final problem

The methods which will be examined here use essentially techniques and algorithms deriving from the famous simplex method, of G.B. Dantzig. This well-known method will not be described in detail (for details see [Dan63, Sim62] or other books). We just recall some terminology that will be used throughout this section.

The aim of the simplex method is to solve the following optimization problem:

$$\begin{array}{c} \sum_{j=1}^{n} c_j x_j = z \\ \min z = z^0 \\ \sum_{j=1}^{n} a_{ij} x_j \leq d_i, \quad i \in \{1, 2, \ldots m\} \\ x_j \geq 0, \quad j \in \{1, 2, \ldots n\} \end{array}$$

for real variables x_i.

The primal simplex algorithm can be applied when the d_i are non-negative. The dual simplex algorithm can be applied when the c_j are non-negative. Dummy variables, one per inequality, are introduced and inequalities are then referred to by the associated variable. For instance, at the first step, we have

$$\begin{array}{rrrrcr} z = & c_1 x_1 & +c_2 x_2 & +\ldots & - & 0 \\ x_{n+i}: & a_{i1} x_1 & +a_{i2} x_2 & +\ldots & \leq & d_i \end{array}$$

Variables appearing in the economic function z are referred to as *out basis variables.* Variables which reference inequalities are called *basis variables.*The simplex is based on the observation

that if the coefficients c_j in z are non-negative and if the coefficients d_i are non-negative, then the problem is solved by giving the value 0 to every out basis variable and the value d_i to the basis variable x_{n+i}. The goal is therefore to obtain such a situation by transformations of the system.

The basic transformation used is a Gaussian elimination algorithm that takes a basis variable out of the basis and inserts an out-basis variable into the basis. Pivoting on (x_k, x_{n+r}) (x_k out of the basis and x_{n+r} in the basis) means replacing x_k by $1/a_{rk}(d_r - x_{n+r} - \sum_{j=1, j\neq k}^{n} a_{rj}x_j)$ in the whole system. Inequality x_{n+r} is renamed x_k. The coefficient a_{rk} is called the *pivot*. The pivot must be chosen such that, in the primal algorithm, the right-hand sides (d_i) remain non-negative and in the dual algorithm the coefficients c_j remain non-negative. Variants of the simplex-based methods differ essentially in the strategy used for choosing the pivot.

Now when the system to solve constrains variables to be integer, simplex method must be adapted, this is explained in the next sections. It should be noted that integer programming specific techniques are not recent. They were essentially developed by R.E. Gomory and are described in classic books such that ([GN72],[SM89]...).

4.1 The Rudimentary Primal All-Integer Algorithm (R.P.A.I.)

This algorithm is a tool used extensively in methods presented in sections 4.2 and 4.3. Its aim is to solve the problem:

$$\begin{array}{c} \sum_{j=1}^{n} c_j x_j = z \\ \min z = z^0 \\ \sum_{j=1}^{n} a_{ij} x_j \leq d_i, \quad i \in \{1, 2, \ldots m\} \\ x_j \in Z, x_j \geq 0, \quad j \in \{1, 2, \ldots n\} \end{array}$$

We assume that all input data are integer and that the right-hand sides d_i are non-negative.

The principle of the algorithm is very close to that of the algorithm for real variables described above. The criteria for pivot selection, optimality, and infinitude are the same. The integer algorithm differs on these points:

- it is clear that when the chosen pivot a_{rk} is ± 1 (more exactly 1 since the pivot is positive), then a Gaussian elimination can be performed and the coefficients of the resulting system are still integers. Now, when the pivot choice criterion indicates a pivot with a value other than 1, from the inequality including it:

  $$\sum_{j=1}^{n} a_{rj} x_j \leq d_r$$

  a new (integer) inequality is constructed, referred to as a "cut" [2]:

  $$\sum_{j=1}^{n} \lfloor a_{rj}/|a_{rk}| \rfloor x_j \leq \lfloor d_r/|a_{rk}| \rfloor$$

  This new inequality is added to the system, and the element located on column k, whose coefficient is now 1 (and is also a possible pivot according to simplex criteria) is chosen as the pivot.

[2] $\lfloor x \rfloor$ stands for the greatest integer less than or equal to x

- Every time a "cut" variable returns to the basis after a pivoting operation, the inequality relative to this cut variable is removed from the system (this point is not strictly obligatory).

Let us consider the problem

$$\begin{array}{rrrrcr} z = & 2x_1 & +2x_2 & -5x_3 & - & 0 \\ x_4: & -3x_1 & -2x_2 & +x_3 & \leq & 3 \\ x_5: & -2x_1 & +3x_2 & -3x_3 & \leq & 4 \\ x_6: & 4x_1 & -x_2 & +2x_3 & \leq & 3 \end{array}$$

The criterion designates the variable x_3 and the inequality x_6 as the coordinates of the pivot. The value of the coefficient a_{rk} is 2, so a cut is added:

$$x_7: \quad 2x_1 \quad -x_2 \quad +x_3 \quad \leq \quad 1$$

and we can execute the pivoting operation (x_3, x_7):

$$\begin{array}{rrrrcr} z = & 12x_1 & -3x_2 & +5x_7 & - & 5 \\ x_4: & -5x_1 & -x_2 & -x_7 & \leq & 2 \\ x_5: & 4x_1 & & +3x_7 & \leq & 7 \\ x_6: & & +x_2 & -2x_7 & \leq & 1 \\ x_3: & 2x_1 & -x_2 & +x_7 & \leq & 1 \end{array}$$

In the next step the set (x_2, x_6) is selected. Since the coefficient a_{rk} is 1, we can pivot directly:

$$\begin{array}{rrrrcr} z = & 12x_1 & +3x_6 & -x_7 & - & 8 \\ x_4: & -5x_1 & +x_6 & -3x_7 & \leq & 3 \\ x_5: & 4x_1 & & +3x_7 & \leq & 7 \\ x_2: & & +x_6 & -2x_7 & \leq & 1 \\ x_3: & 2x_1 & +x_6 & -x_7 & \leq & 2 \end{array}$$

Now, the same criterion selects the set (x_7, x_5). A cut of inequality x_5 is needed since $a_{rk} = 3$:

$$x_8: \quad x_1 \quad +x_7 \quad \leq \quad 2$$

We pivot a third time with the set (x_7, x_8):

$$\begin{array}{rrrrcr} z = & 13x_1 & +3x_6 & +x_8 & - & 10 \\ x_4: & -2x_1 & +x_6 & +3x_8 & \leq & 9 \\ x_5: & x_1 & & -3x_8 & \leq & 1 \\ x_2: & 2x_1 & +x_6 & +2x_8 & \leq & 5 \\ x_3: & 3x_1 & +x_6 & +x_8 & \leq & 4 \\ x_7: & x_1 & & +x_8 & \leq & 2 \end{array}$$

The cut inequality x_7 can be removed, but this is unnecessary. Since the coefficients of the function z are greater than or equal to 0, the problem is finished. Minimum is $z^0 = -10$ with the variables:

$$x_1 = 0 \quad x_2 = 5 \quad x_3 = 4$$

Comments

This algorithm is "all-integer" since the coefficients of the transformed systems all remain integer. The fact that only integer arithmetic is required is an attractive feature of this algorithm, unlike real simplex methods where fractional arithmetic has to be used for avoiding rounding errors.

Nevertheless, it is a rudimentary algorithm. Its completion is not guaranteed. Some rules may be added to guarantee that the algorithm be finite, but, as [NW88] says of that approach: "unfortunately, it is not a practical algorithm because it tends to require an exorbitant number of cuts". However, since most problems which occur in data dependence analysis are small, this algorithm may offer some advantages.

4.2 Constraint-Matrix Test

This method, described in [Wal88], can be applied if the system is composed of m_1 inequalities and m_2 equations such that the variables are constrained and the right-hand sides of inequalities are non-negative:

$$\begin{array}{ll} \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, \ldots m_1\}, \quad d_i \geq 0 \\ \sum_{j=1}^{n} a_{ij}x_j = d_i, & i \in \{m_1+1, \ldots m_1+m_2\} \\ x_j \geq 0 & \end{array}$$

The main principle of the method is the following: at every step, an equivalent and analogous system is built, but with one equation less (and one variable less). The number of inequalities may increase.

If a system without equations or with only equations with right-hand sides equal to zero is obtained, then the system has an obvious solution (all x_j equal to zero). Let us describe the method.

1. An equation is chosen, for instance the first one ($l = m_1 + 1$). We assume that $d_l \geq 0$. If not, we change the sign of every coefficient in the equation.

2. If $d_l > 0$ we consider the subsystem:

$$\begin{array}{ll} \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, \ldots m_1\} \\ \sum_{j=1}^{n} a_{lj}x_j = d_l & \end{array}$$

   We define the economic linear function z:

$$z = d_l - \sum_{j=1}^{n} a_{lj}x_j$$

   and the corresponding optimization problem:

$$\begin{array}{ll} \min z = z^0 & \\ \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, \ldots m_1\} \\ \sum_{j=1}^{n} a_{lj}x_j \leq d_l & \end{array}$$

(Note that the function and the last constraint are analogous)

We solve this problem by the Rudimentary Primal All-Integer Algorithm (4.1). The solution z^0 is necessarily finite and non-negative.

- If the value of z^0 is positive, the subsystem has no solution, hence original system has no solution either.
- Otherwise z^0 is zero, indicating that the subsystem has a solution. If there is no other equations, the full system has one also.

3. The full system is now of the form (for the sake, we use the notations of the initial system, but note that introducing cuts may have increased the number of inequalities):

$$\begin{array}{ll} \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, \ldots m_1\} \quad (d_i \geq 0) \\ \sum_{j=1}^{n} a_{ij}x_j = 0, & i = m_1 + 1 \\ \sum_{j=1}^{n} a_{ij}x_j = d_i, & i \in \{m_1 + 2, \ldots m_1 + m_2\} \\ x_j \geq 0 & \end{array}$$

Our purpose is now to eliminate the first equation (while keeping non-negative right-hand sides of the inequalities). This is easy in the following cases:

all the coefficients of the equation have the same sign (variables with non-zero coefficients are necessarily equal to zero and disappear like the equation from the system)

the equation has an unitary coefficient (a Gaussian elimination is performed and the equation is replaced by an inequality, see the note below).

Otherwise, a cut (analogous to those used in the Rudimentary Primal All-Integer Algorithm) followed by a pivoting operation are performed repeatedly, until an opportunity for elimination appears in the equation.

Note

When a Gaussian elimination step removes a constrained variable from the system, the equation must be replaced by an inequality indicating that the eliminated variable was constrained. For instance, we can eliminate x_2 if the system includes an equation such as:

$$-2x_{10} \quad +x_2 \quad +x_9 \quad = \quad 0$$

but since $x_2 \geq 0$, we must introduce the inequality:

$$-2x_{10} \quad +x_9 \quad \leq \quad 0$$

4.3 Method "FAS3T"

This algorithm [BP89] assumes that the system is composed of inequalities:

$$\sum_{j=1}^{n} a_{ij}x_j \leq d_i, \quad i \in \{1, 2, \ldots m\}$$

The principle of the method does not depend on whether or not the variables are constrained, so it will be explained without reference to this point.

The algorithm starts with a point X^1 (a vector with components x_j). At every step, k, of the method, we have a current point X^k and two sets of constraints, constraints that are verified by X^k and constraints that are not.

$$I_1 = \{i \ \ /A_i X^k \leq d_i\}$$
$$I_2 = \{i \ \ /A_i X^k > d_i\}$$

If the set I_2 is empty then obviously X^k is a solution.

Otherwise, a function z and a minimization problem are defined:

$$z = \sum_{i \in I_2} A_i X$$
$$\min z = z^0$$
$$A_i X \leq d_i \ \ i \in I_1$$

X^{k+1} is defined as the point where z is minimized.

It can be proven that the original problem has no solution in one of the following cases:

1. $z^0 > \sum_{i \in I_2} d_i$
2. $z^0 = \sum_{i \in I_2} d_i$ and $\exists i \in I_2 \ \ /A_i X^{k+1} \neq d_i$

Otherwise, one or more inequalities, previously in the set I_2, necessarily verify:

$$A_i X^{k+1} \leq d_i$$

and will be incorporated in set I_1 in the next step.

The number of such steps is obviously finite. For the solution of each minimization problem, the system is modified so that variables are constrained. Then the Rudimentary Primal All-Integer Algorithm (section 4.1) is used.

This method requires a finite solution for each current problem. Generally, variables of systems resulting from dependence analysis are bounded, but it may occur that domains relative to the generic problems used by the method are not. In these cases, we are not guaranteed that finite solutions will be obtained. If necessary, constraints limiting the domain will be added in such a way that they do not modify the result of the algorithm (for example giving to variables bounds with large values).

4.4 Simple Dual All-Integers Test

This algorithm, proposed in [Sog92], supposes that the system is composed of inequalities with constrained variables:

$$\begin{array}{ll} \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, 2, \ldots m\} \\ x_j \geq 0 & \end{array}$$

For a real simplex algorithm, the dual problem consists roughly in inverting the roles of the minimization function coefficients, c_j, and right-hand sides, d_i.

The principle is very close to the classic finite Dual All-Integers algorithm developed by Ralph Gomory and described in [GN72][SM89], which finds the minimum of a linear function z satisfying:

$$\begin{array}{ll} \min \sum_{j=1}^{n} c_j x_j = z & \\ \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, 2, \ldots m\} \\ x_j \in Z, x_j \geq 0, & j \in \{1, 2, \ldots n\} \end{array}$$

The coefficients c_j must be non-negative (like d_i right-hand sides for the primal algorithm). We note that if we choose a null function z (all c_j equal to $zero$), the problem solved by the finite Dual All-Integers algorithm reduces to ours: is there an integer solution satisfying the inequalities?

Nevertheless, the proposed method does not use the somewhat computationally expensive lexicographic rule mentioned in [GN72] for ensuring the finiteness of the algorithm. Naturally, the choice of the pivot is different from the classic algorithm. Consequently, like the Rudimentary Primal All-Integer Algorithm (4.1), we are not guaranteed that the algorithm is finite.

The steps of the method are very similar to the primal algorithm.

We start with the initial system and we execute the following sequence as many times as necessary:

1.
   - If in the current system all d_i are greater than or equal to zero, an obvious solution exists.
   - If the current system includes an inequality i such that $d_i < 0$ and all coefficients a_{ij} are ≥ 0, the system has no solution.
   - If the number of iterations exceeds a fixed number $tmax$, for instance $3(m+n)$, we stop. We may not conclude anything about the solution.
2. We select the inequality (r) with a negative d_i (best results are obtained when d_i has the largest absolute value).

   $$\sum_{j=1}^{n} a_{rj}x_j \leq d_r$$

   For the variable x_k, we choose a variable in the selected inequality that has a negative coefficient. Our choice was the a_{rj} with the largest absolute value.
3.
   - If the coefficient a_{rk} is equal to -1, it is chosen as pivot element.

- Otherwise, the following cut is added to the current system of inequalities:

$$\sum_{j=1}^{n} \lfloor a_{rj}/|a_{rk}| \rfloor x_j \leq \lfloor d_r/|a_{rk}| \rfloor$$

  and the coefficient a_{sk} (equal to -1) of the new inequality (s) is chosen as pivot element (note that the cutting rule is the same than that used in the Rudimentary Primal All-Integer Algorithm (4.1)).

4. A pivoting operation is performed.

5. If the variable x_k is a "cut variable" (i.e. if x_k appeared in connection with a previous cutting operation), the corresponding inequality is removed from the current system.

For example, let us consider the system with constrained variables:

$$\begin{array}{rrrcr} -3x_1 & -2x_2 & +x_3 & \leq & 3 \\ -2x_1 & +3x_2 & -3x_3 & \leq & -4 \\ x_1 & -x_2 & +2x_3 & \leq & -3 \\ 2x_1 & +2x_2 & -5x_3 & \leq & -1 \end{array}$$

Using slack variables ≥ 0 this system is written:

$$\begin{array}{lrrrcr} x_4: & -3x_1 & -2x_2 & +x_3 & \leq & 3 \\ x_5: & -2x_1 & +3x_2 & -3x_3 & \leq & -4 \\ x_6: & x_1 & -x_2 & +2x_3 & \leq & -3 \\ x_7: & 2x_1 & +2x_2 & -5x_3 & \leq & -1 \end{array}$$

We select the inequality x_5 and the variable x_3. Since value (-3) of the corresponding coefficient is not -1, we add a cut, introducing a cut variable x_8 constructed from x_5:

$$\begin{array}{lrrrcr} x_4: & -3x_1 & -2x_2 & +x_3 & \leq & 3 \\ x_5: & -2x_1 & +3x_2 & -3x_3 & \leq & -4 \\ x_6: & x_1 & -x_2 & +2x_3 & \leq & -3 \\ x_7: & 2x_1 & +2x_2 & -5x_3 & \leq & -1 \\ x_8: & -x_1 & +x_2 & -x_3 & \leq & -2 \end{array}$$

A pivoting operation is executed (variables x_8, x_3):

$$\begin{array}{lrrrcr} x_4: & -4x_1 & -x_2 & +x_8 & \leq & 1 \\ x_5: & x_1 & & -3x_8 & \leq & 2 \\ x_6: & -x_1 & +x_2 & +2x_8 & \leq & -7 \\ x_7: & 7x_1 & -3x_2 & -5x_8 & \leq & 9 \\ x_3: & x_1 & -x_2 & -x_8 & \leq & 2 \end{array}$$

Next, we select the inequality x_6 and the variable x_1. This time, no cut is necessary and we can execute the pivoting operation (x_6, x_1):

$$\begin{array}{lrrrcr} x_4: & -4x_6 & -5x_2 & -7x_8 & \leq & 29 \\ x_5: & x_6 & +x_2 & -x_8 & \leq & -5 \\ x_1: & -x_6 & -x_2 & -2x_8 & \leq & 7 \\ x_7: & 7x_6 & +4x_2 & +9x_8 & \leq & -40 \\ x_3: & x_6 & & +x_8 & \leq & -5 \end{array}$$

The inequalities x_7 and x_3 cannot be true, since the variables must be non-negative. Hence the system has no solution.

Comments

Like the Rudimentary Primal All-Integer Algorithm (4.1)), this algorithm is attractive since it is all integer. Like the R.P.A.I., it is not a finite algorithm. However, general behavior of non-finite algorithms is often as good or better than that of finite algorithms (because in practice, the expected number of iterations in the finite algorithm is very large. An example is the classic simplex method). Its behavior is probably comparable to that of the finite Dual All-Integers algorithm, but there are few precise comments about the behavior of the finite algorithm in literature.

Experiments seemed interesting. Some problems with 9 variables, for instance, could be solved easily while the same problems were solved with great difficulty by methods using the R.P.A.I. (we must specify that these systems were not issued from data dependence analysis, but from more complex task scheduling problems). However, the algorithm seems sensitive to the choice of the pivot (many choices are possible). Some artificial problems could be solved with very few iterations, but required much more iterations or could not terminate with a different choice of pivot. But this inconvenience was not noticed with problems coming from data dependence analysis (they are very "easy").

4.5 Surrogate Dual All-Integers Test

This algorithm ([Sog92]) is a variant of the previous one and supposes that the system is composed of inequalities with constrained variables:

$$\begin{array}{ll} \sum_{j=1}^{n} a_{ij}x_j \leq d_i, & i \in \{1, 2, \ldots m\} \\ x_j \geq 0 & \end{array}$$

The principle is similar to the Simple Dual All-Integers Test. We start with the initial system and we execute an analogous sequence of operations as many times as necessary. The sequence differs on these points:

- If the number of inequalities with a negative right-hand side is greater than 1, a surrogate constraint is built, which is the sum of all such inequalities. A similar rule for selecting a pivot is applied to this new inequality. If no negative pivot is found,

the system has no solution. If the value of the pivot obtained is -1, the surrogate inequality is added to the system (introducing a "surrogate" slack variable), otherwise a cut is built from the surrogate inequality and added to the system. Then, the usual pivoting operation is executed,

- If a "surrogate" variable is returned to the basis, it is removed from the current system (like a "cut" variable).

Let us return to the previous problem (4.4):

$$\begin{array}{rrrrcr} x_4: & -3x_1 & -2x_2 & +x_3 & \leq & 3 \\ x_5: & -2x_1 & +3x_2 & -3x_3 & \leq & -4 \\ x_6: & x_1 & -x_2 & +2x_3 & \leq & -3 \\ x_7: & 2x_1 & +2x_2 & -5x_3 & \leq & -1 \end{array}$$

Three of the inequalities have a negative right-hand side. Their sum gives another inequality (y denotes a temporary variable):

$$y: \quad x_1 \quad +4x_2 \quad -6x_3 \quad \leq \quad -8$$

The coefficient of x_3 is selected, but its value is -6. A cut x_8 is built from y and added to the system:

$$x_8: \qquad -x_3 \quad \leq \quad -2$$

A pivoting operation is executed (x_8, x_3):

$$\begin{array}{rrrrcr} x_4: & -3x_1 & -2x_2 & +x_8 & \leq & 1 \\ x_5: & -2x_1 & +3x_2 & -3x_8 & \leq & 2 \\ x_6: & x_1 & -x_2 & +2x_8 & \leq & -7 \\ x_7: & 2x_1 & 2x_2 & -5x_8 & \leq & 9 \\ x_3: & & & -x_8 & \leq & 2 \end{array}$$

The set (x_6, x_2) is selected. No cut is necessary and we can pivot:

$$\begin{array}{rrrrcr} x_4: & -5x_1 & -2x_6 & -3x_8 & \leq & 15 \\ x_5: & x_1 & +3x_6 & +3x_8 & \leq & -19 \\ x_1: & -x_1 & -x_6 & -2x_8 & \leq & 7 \\ x_7: & 4x_1 & +2x_6 & -x_8 & \leq & -5 \\ x_3: & & & -x_8 & \leq & -2 \end{array}$$

There is no solution to the inequality x_5. Hence the system has no solution.

Comments

With most of the problems which were tested, the behavior of this algorithm was generally analogous to that of the Simple Dual All-Integers Test. However long sequences of iterations were not observed with the Surrogate Test, which seems not very sensitive to the choice of the pivot.

	Trivial	Gauss (a)	GCD test	Gauss (b)	Var. Chang.	Fourier-Motzkin	Non-neg. Var.	Dual All Int.
NONLINEAR	18.5 %	0.6 %	0.3 %	0 %	0 %	66.1 %	13.9 %	0.6 %
LINEAR	7 %	0.7 %	0.6 %	0.003 %	0 %	74.7 %	14 %	3 %

Table 1: Case of exit

	Tot. Numb.	Gauss (a)	GCD test	Gauss (b)	Var. Ch..	Four.-Motz	Non-neg. Var.	Dual All Int.
Aver.	4.9 - 5.7	1.5 - 1.3	0.003 - 0.006	0.004 - 0.018	0 - 0	3.2 - 4.2	0.2 - 0.25	0.006 - 0.035
Max.	17 - 18	6 - 4	1 - 1	1 - 1	0 - 0	14 - 15	3 - 5	3 - 7

Table 2: Number of each step's iterations per dependence test

5 Which strategy?

A strategy must now be defined for chaining the different steps described above, so as to minimize the sum of the costs of the reduction phase and the resolution phase.

Let us first analyze the cost of each step. The cost of a reduction step can be easily estimated: an equation elimination requires a possible change of variables and a Gaussian elimination (technically a matrix pivoting operation). Since each time an equation is eliminated, the size of the matrix decreases, the cost of subsequent reduction operations decreases. A variable elimination by selective Fourier-Motzkin elimination under the conditions described in section 3.7 is not very expensive either, because the basic operation is rewriting the system. The number of resulting inequalities does not increase, while the number of variables decreases.

Estimating the average cost of what we call the problem resolution phase is generally not so easy. Whatever method is used, there are "easy" problems that are solved with two or three iterations while some others require more than ten iterations. We can only measure the cost of a single iteration: a matrix pivoting operation, possibly preceded by the introduction of a "cut" inequality, which increases the size of the system.

Therefore, the conclusion is that reductions steps are cheap while resolution steps may be expensive.

Let us now compare the integer programming methods presented in section 4. The Constraint matrix test needs as many integer simplex solutions as the number of equations in the system. The size of the induced simplex problems is the current number of inequalities plus one. The FAS3T method needs as many integer simplex solutions as the number of inequalities in the worst case, and the size of simplex subproblems grows roughly from one to the total number of inequalities. The Dual all integer algorithm solves only one integer simplex, the size of which is the number of inequalities. Since the cost of one integer simplex solution is unpredictable, we think it is worthwhile to use as few such steps as possible. That is why we recommend the following strategy:

1. eliminate all equalities:

   (a) Use Gaussian elimination for equations including unitary coefficients (step Gauss (a)).

(b) Apply the GCD test to each remaining equality. If it fails, divide the equation by its GCD. If no unitary coefficient appears, generate one by changing variables (3.4, 3.5). Then use a Gaussian elimination (step Gauss (b)).

2. eliminate some remaining variables:

   (a) Divide all inequalities by their GCD.

   (b) Use as many selective Fourier-Motzkin elimination steps as possible.

3. Introduce constrained variables:

   (a) Change by slack variables as many (free) variables as possible.

   (b) If some free variables remain, use the "single added variable" technique.

4. Use the surrogate (or other) dual all-integer algorithm.

Variations in the order of the application of reduction steps can be imagined, since it may be possible to apply more than one method at a given step. However, we think it is fundamental to eliminate equations as soon as possible.

We have implemented this strategy in PIAF parallelizer developed at INRIA [GLL+90]. For evaluating its performance, we have used as input data systems issued from Perfect Club Benchmark [BCK+89]. About 35,000 data dependence tests have been performed. For multi-dimensional arrays, two cases have been considered. In the first case (NONLINEAR), they were treated dimension by dimension (possibly leading to more than one equation in the system). In the second case (LINEAR), accesses have been linearized (therefore the systems have generally only one equation).

Table 1 reports after which step the algorithm was able to conclude to the existence or not of a data dependence. In most cases, the answer was given after the step of Fourier-Motzkin elimination. In many cases (especially in NONLINEAR cases, 18.5 %) a trivial answer was given during step Gauss (a) (case where an equality has no solution since each of its coefficients is zero while its right-hand side is non-zero). In only a few cases (0.6 % and 3 %), the Dual All Integer algorithm was necessary for conclusion, essentially because the step of introducing non-negative variables directly gave the answer (13.9 % and 14% of cases). This reinforces the idea that reduction steps are very important and must not be neglected in a general integer linear programming method.

The table 2 gives indications about the cost of our method: in each case, the two numbers correspond respectively to NONLINEAR and LINEAR case. The first line gives the average of iterations necessary for conclusion, for each step, while the second line gives the maximum number of iterations used. In average, the result is roughly that 5 (6) iterations are necessary for each step, among which 1.5 (1.3) iteration for (trivial) Gaussian elimination and 3.2 (4.2) iterations for Fourier-Motzkin elimination are performed. The maximum numbers of iterations in each step are not very large either, thus no infinite behavior is observed. Last, the maximum number of total iterations found is small (17 and 18). Since the price for one iteration is about one variable change, the conclusion is that our exact method is very cheap.

6 Conclusion

In this paper we have presented a new algorithm for data dependence analysis. It is implemented in INRIA PIAF parallelizer [GLL+90]. Its first quality is its accuracy, this is an exact algorithm. Next, it applies to any integer linear programming satisfiability problem and therefore can handle non-standard data informations resulting from sophisticated data flow analysis methods. More, it can be applied to other phases in a compiler. For now it is used also for variable compatibility verifications, but we plan to use it also for solving a scheduling problem for exploiting fine grain parallellism. Last but not least, this test remains very efficient thanks to a powerful reducing preprocessing phase that makes the problem grandly simplified. The measurements performed in PIAF on Perfect Club Benchmark show that the price to pay for using an exact method is almost insignificant. Therefore data dependence analysis research can turn now towards more sophisticated data-flow analysis techniques.

Acknowledgments

The authors wish to thank PIAF's main author François THOMASSET for his help in using the PIAF parallelizer, Dannie DURAND for her help in coping with english language, and anonymous referees for their valuable comments.

References

[Ban79] Uptal Banerjee. *Speedup of ordinary programs.* PhD thesis, University of Illinois, 1979.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, Mass, 1988.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The PERFECT Club benchmarks: effective performance evaluation of supercomputers. CSRD Report 827, University of Illinois, Urbana-Champaign, May 1989.

[BP89] H. Bennaceur and G. Plateau. Sur le problème de satisfaction de contraintes. Rapport LIPN 89- 6, Universite Paris Nord, Centre Scientifique et Polytechnique, Département de mathématiques et informatique, Juin 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the Forth ACM Conference on Principles of Programming Languages*, 1977.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the Fifth ACM Conference on Principles of Programming Languages*, 1978.

[Dan63] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, New-Jersey, 1963.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In SIGPLAN Notices, editor, *Proc. of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26, pages 15–29, Toronto, Ontario, Canada, June 26-28 1991. ACM.

[GLL+90] Marie Christine Giboulot, Eve Rose Lebon, Michel Loyer, Gregory Popovitch, Husein Shafie, and François Thomasset. Parallel execution of Fortran programs on the EWS workstation. In Roland Glowinski and Alain Lichnewsky, editors, *Computing Methods in Applied Sciences and Engineering*, pages 413–423, Philadelphia, January 1990. SIAM.

[GN72] R. S. Garfinkel and G. N. Nemhauser. *Integer Programming.* Wiley-Interscience, 1972.

[Gra89] Philippe Granger. Static analysis of arithmetic congruences. *Int J Computer Math*, 30:165–190, 1989.

[KKP91] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I test: an improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, July 1991.

[Knu81] D. E. Knuth. *The Art of Computer Programming, Vol 2, Seminumerical Algorithms.* Second Edition, Addison-Wesley, Reading, Massachusetts, 1981.

[KPK90] David Klappholz, Kleanthis Psarris, and Xiangyun Kong. On the perfect accuracy of an approximate subscript analysis test. In *Proceedings of ACM 1990 International Conference on Supercomputing*, Amsterdam, Holland, June 1990.

[Len83] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[LT88] Alain Lichnewsky and François Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *Proceedings of 1988 International Conference on Supercomputing*, Saint-Malo, France, July 1988.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In SIGPLAN Notices, editor, *Proc. of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26, pages 1–14, Toronto, Ontario, Canada, June 26-28 1991. ACM.

[Min83] M. Minoux. *Programmation mathématique, théorie et algorithmes, tomes 1 et 2.* Dunod, 1983.

[NW88] G. N. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, 1988.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience, 1986.

[Sim62] B. Simonnard. *Programmation linéaire.* Dunod, Paris, 1962.

[SM89] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming.* North-Holland, 1989.

[Sog92] Jean-Claude Sogno. Analysis of standard and new algorithms for the integer and linear constraint satisfaction problem. Rapport de recherche, INRIA, 1992. to appear.

[Wal88] D. R. Wallace. Dependence of multi-dimensional array references. In *Proceedings of 1988 International Conference on Supercomputing*, Saint-Malo, France, July 1988.

[Wol82] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1982.

[WZ85] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Proc. of the 12th ACM Conference on Principles of Programming Languages*, pages 291–299, 1985.