Analyse statique de programme et preuve automatique : étude des problèmes rencontrés avec la bibliothèque linéaire de PIPS

Duong NGUYEN QUE

Rapport du CRI-ENSMP

De 15 Mars 2002 a 31 Aout 2002

8 octobre 2002

Résumé

Les analyses statiques de programmes nécessitent un mécanisme de preuve automatique. Le mécanisme utilisé dans l'outil PIPS est fondé sur une abstraction des ensembles par des polyèdres convexes.

En manipulant ces polyèdres convexes, nous sommes rencontrés plusieurs problèmes concernant les complexités en temps, espace et magnitude. Le but est de comprendre pourquoi certains algorithmes sur des cas, relativement simples au départ, finissent par diverger et générer des overflows. Dans les limites d'un stage de fin d'études, nous avons étudié et puis proposé une solution pour le calcul de la satisfiabilité. Nous aimerons traiter d'autres problèmes dans le futur.

Remerciements

Ce stage a eu lieu au Centre de Recherche en Informatique de l'Ecole Nationale Supérieure des Mines de Paris (CRI-ENSMP), France.

Je voudrais d'abord remercier Corinne Ancourt, mon responsable de stage, pour m'avoir beaucoup aidé dès mon arrivée en France jusqu'à mon départ. Je tiens à remercier François Irigoin, qui m'a initié à la recherche dans le domaine de la programmation linéaire. Ses suggestions m'ont orienté efficacement et ont permis l'aboutissement de ce travail. Je suis aussi très reconnaissant à Fabien Coelho qui m'a fourni le support nécessaire pour résoudre des problèmes rencontrés. Je remercie Georges-André Silber pour avoir patiemment lu et corrigé mon rapport.

Je remercie également le Professeur Robert Mahl, directeur du Centre de Recherche en Informatique de l'Ecole Nationale Supérieure des Mines de Paris, pour m'avoir chaleureusement accueilli dans son laboratoire.

Je voudrais également remercier le Professeur Paul Feautrier et Monsieur Jean-Claude Sogno, pour m'avoir aidé dans les études des systèmes de contraintes, et surtout pour m'avoir fourni leurs logiciels avec les codes sources les plus récents.

Mes remerciements vont également à l'ensemble des membres du Centre de Recherche en Informatique où ce stage a été préparé dans une ambiance très agréable avec les meilleures conditions, malgré les limites d'un stage.

Table des matières

1 Introduction **9**
1.1 Contexte actuel 9
1.2 Objectifs 10
1.3 Organisation du rapport 10

2 État de l'art **13**
2.1 Polyèdre convexe 13
2.2 Opérations de base 14
2.2.1 Satisfiabilité 14
2.2.2 Projection 15
2.2.3 Normalisation 15
2.2.4 Intersection, union et enveloppe convexe 16
2.3 Ocean, problèmes rencontrés 16
2.4 Travaux antérieurs - Janus et PIP 17

3 Méthodologie **19**
3.1 Constitution d'une base de tests 19
3.2 Instrumentations 20
3.2.1 Impression d'un système de contraintes 20
3.2.2 Exception time-out 20
3.2.3 Changement de méthode en cas d'échec 22
3.3 D'autres implantations 22
3.3.1 Problème de magnitude - un essai avec preprocess 22
3.3.2 Problème de l'ordre de contraintes 22
3.3.3 Convertisseurs 24

4 Satisfiabilité **25**
4.1 Satisfiabilité dans LINEAR 25
4.1.1 Méthode du Simplexe 25

4.1.2 Méthode de Fourier-Motzkin 27
4.2 Problèmes 27
4.2.1 Heuristique entre Simplexe et Fourier-Motzkin dans LINEAR 27
4.2.2 Vitesse et espace mémoire 28
4.2.3 Overflow et exception 28
4.3 Comparaisons avec Janus et PIP64 29
4.3.1 Critère de pivotage 29
4.3.2 Critère de temps 29
4.3.3 Critère d'espace mémoire 34
4.4 Intégration de Janus dans PIPS 35

5 Conclusions **37**

Chapitre 1

Introduction

1.1 Contexte actuel

Il s'agit d'un outil d'analyse, de transformation et d'instrumentation de programmes appelé PIPS (Paralléliseur Interprocédural de Programmes Scientifiques) [?], qui est utilisé pour étudier de nombreux programmes de "benchmarks" publics et pour évaluer des applications industrielles. L'implantation existante et son utilisation permettent d'évaluer l'ampleur des problèmes posés par l'utilisation d'un prouveur automatique dans un contexte de compilation.

En fait, la parallélisation automatique, qui consiste à transformer un programme séquentiel exprimé dans un langage de haut niveau d'une part une analyse des dépendances pour exposer le parallélisme potentiel, et d'autre part des transformations de programme comme la privatisation de tableaux pour accroître la localité des références.

L'équipe d'analyse, de transformation et d'instrumentation de programmes du centre, dans le logiciel PIPS, a utilisé des algorithmes manipulant des contraintes affines comme système de preuve pour obtenir automatiquement le résultat d'une analyse, pour prouver la correction d'une transformation de programme ou pour générer un code équivalent à une spécification.

L'utilisation d'un tel système basé sur les techniques d'algèbre linéaire, est effectuée par un ensemble des libraires écrites en C, qui s'appelle C3 (LINEAR). Deux types de problèmes se posent lors de l'utilisation du système :

1. Les complexités en temps, espace et magnitude.
2. L'adéquation des modélisations sous forme de contraintes affines aux objectifs d'analyses.

Pour le deuxième type, il est possible qu'une explosion combinatoire d'une seule preuve élémentaire parmi des milliers bloque une analyse automatique ou bien en multiplie le temps d'exécution par un facteur supérieur à dix.

1.2 Objectifs

Le but de ce stage est, dans un premier temps, de comprendre pourquoi certains algorithmes sur des cas, relativement simples au départ, finissent par diverger et générer des overflows. Une base de données de cas limites rencontrés par les algorithmes doit être constituée. Dans un deuxième temps, des heuristiques et nouveaux algorithmes doivent être proposés pour améliorer la gestion de ces overflows.

Dans la bibliothèque LINEAR, des algorithmes d'algèbre linéaire sont utilisés afin de prouver des propriétés des programmes à analyser. En réalité, dans la plupart de cas, les analyses des programmes nous donnent des systèmes de contraintes dont la satisfiabilité est à vérifier.

Nous utilisons dans ce rapport le terme "satisfiabilité", plutôt que "faisabilité", même si ce terme n'existe pas en français. Nous définissons ici le terme comme le fait de satisfaire toutes les contraintes dans un système. Des algorithmes de la programmation linéaire comme le Simplexe, Fourier-Motzkin, ou bien PIP (programmation intégrale paramétrique) sont implémentés ou intégrés dans LINEAR, et largement utilisés par PIPS.

En outre, la projection et la normalisation des systèmes de contraintes sont des opérations qui peuvent causer également des problèmes. Le calcul d'une enveloppe convexe de deux systèmes de contraintes est un grand consommateur du temps d'exécution en cas d'explosion, où il y a probablement des erreurs.

Notre motivation est de construire des systèmes de preuve sûrs, automatiques et rapides. Plus concrètement, nous souhaitons de trouver de meilleures solutions qui pourront réduire le nombre d'overflows et améliorer la vitesse des calculs dans tous les cas. Dans ce stage, nous nous concentrons sur les algorithmes permettant de vérifier la satisfiabilité d'un système de contraintes affines. Les autres problèmes seront étudiés dans l'avenir.

1.3 Organisation du rapport

D'abord, dans le chapitre **??**, nous décrivons le contexte actuel qui contient des opérations de base dans la bibliothèque LINEAR, ainsi que les problèmes rencontrés. Ensuite nous présentons les travaux antérieurs et diverses méthodes calculant la satisfiabilité d'un système de contraintes. Dans la description des opérations de base, nous donnons quelques définitions importantes pour qu'on puisse mieux comprendre l'environnement du logiciel PIPS.

Les travaux antérieurs et les résultats de la recherche bibliographique sont rapidement présentés, y compris d'autres logiciels avec d'autres algorithmes, ou seulement d'autres implantations qui sont différentes en comparaison avec celles qui sont dans LINEAR. Puis, nous allons étudier les méthodes de façon théorique.

Ensuite, le chapitre **??** explique comment nous attaquons les problèmes. Nous parlons également des instrumentations intégrées dans LINEAR pour faciliter les travaux, et d'autres implantations diverses.

Le chapitre **??** est le plus important. Il se concentre en détails sur la satisfiabilité des systèmes de contraintes. C'est là que se trouve la comparaison entre les implantations dans LINEAR, Janus, et PIP avec des expérimentations et des tests. Une intégration du Janus dans LINEAR et quelques tests sont présenté.

Enfin, dans le chapitre **??**, se trouvent la conclusion et les perspectives.

Chapitre 2

État de l'art

2.1 Polyèdre convexe

Au cours de ces dix dernières années, de nombreuses études ont été consacrées à la représentation des ensembles des valeurs qu'une variable peut recevoir en exécutant un programme. La complexité en temps et en espace mémoire, ainsi que la précision des résultats, en dépendent. Tout l'art consiste à trouver un équilibre satisfaisant entre ces paramètres.

Dans PIPS, il a été choisi d'utiliser la convexité pour réduire la demande en espace mémoire, tout en conservant une bonne précision. Les ensembles sont donc représentés par des polyèdres convexes paramétrés. Nous vous présentons les quelques définitions suivantes :

Définition 2.1.1 *Un polyèdre convexe en nombres entiers Z est un ensemble de points de Z^n défini par :*

$$P = \{x \in Z^n : A.x \leq b\}$$

où A est une matrice de $Z^m \times Z^n$ et b un vecteur de Z^m.

P peut être vu comme l'intersection des demi-espaces définis par ses contraintes. Un polyèdre peut aussi être représenté par son *système générateur* défini comme suit :

Définition 2.1.2 *Soit P un polyèdre convexe. Son système générateur est constitué de trois ensembles : un ensemble de sommets $V = \{v_1, \ldots, v_\sigma\}$, un ensemble de rayons $R = \{r_1, \ldots, r_\rho\}$, et un ensemble de droites $L = \{l_1, \ldots, l_\delta\}$.*

1. Un *sommet* d'un polyèdre P est un point $v \in P$ qui n'est pas combinaison linéaire d'autres points de P :

$$\begin{cases} v = \sum_{i=1}^{p} \lambda_i x_i \\ \forall i = 1, \ldots, p : x_i \in P, \lambda_i \geq 0 \\ \sum_{i=1}^{p} \lambda_i = 1 \end{cases} \Rightarrow \ \forall i = 1, \ldots, p : \lambda_i = 0 \ or \ v = x_i$$

2. Un *rayon* d'un polyèdre P est un vecteur $r \in R^n$ tel qu'il existe une demi-droite parallèle à r, et entièrement contenue dans P :

$$\forall x \in P,\ \forall \mu \in R^+,\ x + \mu r \in P$$

3. Une *droite* d'un polyèdre P est un vecteur $l \in R^n$ tel que l et $-l$ sont des rayons de P :

$$\forall x \in P\ \forall \nu \in R\ x + \nu l \in P$$

P est alors défini par :

$$\left\{ x : \left| \begin{array}{l} \exists \lambda_1, \ldots, \lambda_\sigma \in [0,1] : \sum_{i=1}^{\sigma} \lambda_i = 1 \\ \exists \mu_1, \ldots, \mu_\rho \in R^+ \\ \exists \nu_1, \ldots, \nu_\delta \in R \end{array} \right. \quad x = \sum_{i=1}^{\sigma} \lambda_i v_i + \sum_{i=1}^{\rho} \mu_i r_i + \sum_{i=1}^{\delta} \nu_i l_i \right\}$$

Ces deux représentations sont équivalentes (duales), et il y a des algorithmes qui permettent de passer de l'une à l'autre. Certaines opérations comme l'intersection sont plus faciles à réaliser sur les systèmes de contraintes, d'autres, comme l'enveloppe convexe, sur les systèmes générateurs.

Système de contraintes. Un polyèdre convexe dans PIPS est représenté par un système de contraintes. Un système de contraintes lui-même contient une *base* et des *contraintes*. La base est constituées des variables représentée par les axes de l'espace. Les contraintes sont simplement des égalités ou des inégalités sur ces variables.

2.2 Opérations de base

2.2.1 Satisfiabilité

Un système de contraintes est dit satisfiable, ou bien faisable, si nous pouvons trouver un ensemble de valeurs des variables qui satisfont toutes les égalités et inégalités. Au contraire, si nous détectons une contradiction entre des contraintes, nous disons que le système de contraintes est insatisfiable ou bien infaisable.

Un autre cas, c'est que nous n'arrivons pas à montrer le système de contraintes satisfiable, ni insatisfiable. Nous disons que le système de contraintes est indéterminé. C'est le cas qui se présente quand une exception est levée.

Dans LINEAR, deux fonctions vérifiant la satisfiabilité d'un système de contraintes sont implémentées. L'une utilise l'algorithme de Fourier-Motzkin[?], l'autre utilise l'algorithme de Simplexe [?]. Chaque algorithme a ses points forts et faibles, donc nous utilisons un heuristique pour déterminer dans quel cas il faut utiliser l'une ou l'autre.

2.2.2 Projection

La projection selon une variable est une opération importante, qui est utilisée partout, y compris dans le calcul de la satisfiabilité de Fourier-Motzkin. Nous pouvons éliminer une variable, ou bien éliminer une contrainte dans un système de contraintes. Par exemple, le calcul des régions d'une boucle nécessite une projection selon l'indice de boucle, c'est-à-dire l'élimination de celui-ci. Dans ce cas, l'indice de boucle est stocké comme une variable dans le système de contraintes.

L'algorithme de Fourier-Motzkin permet d'effectuer cette élimination. En réalité, il calcule essentiellement l'enveloppe convexe de la projection des points rationnels du polyèdre de départ. Le résultat est donc une sur-approximation de l'ensemble recherché, réduite par une prise en compte du caractère entier des variables lors de la normalisation des contraintes.

Normalement, nous souhaitons de projeter un système de contraintes afin de le simplifier, pour obtenir un système de contraintes équivalent. Dans ce cas, il pose une question : nous devons choisir quelle variable, ou bien quelle direction, à projeter, pour que le système soit plus simple ? Dans LINEAR, plusieurs manières de choisir la variable sont proposées. Malheureusement dans certains cas, après une projection, la combinaison deux-à-deux explose et nous donne de nouveaux systèmes de contraintes ayant un nombre de contraintes très grand.

Cela entraîne des problèmes pour garder le système de contraintes dans la mémoire et le problème de temps d'exécution. De plus, l'explosion sera accumulée, et entraînera des systèmes de contraintes de plus en plus gros. S'il est grave, nous avons un arrêt du programme.

Il faut donc étudier le problème et chercher à comprendre, dans quel cas, avec quelle variable, nous risquons d'avoir une explosion si nous effectuons la projection le long de cette variable. Comme résultat, nous pouvons éviter les explosions indésirables.

2.2.3 Normalisation

La normalisation est une opération au niveau un peu plus haute que la projection. Elle utilise la projection et l'élimination de redondances dans certains cas. Son but est de normaliser partiellement le système de contraintes et aussi de détecter l'insatifiabilité du système.

Dans la normalisation, il y a une question sur la validation : est-ce que la normalisation est correcte ? Et un problème de la robustesse : est-ce qu'on peut fait mieux, quand la normalisation est une opération coûteuse ? En plus, il y a quelques cas où après une normalisation, nous avons une explosion dans le calcul d'une enveloppe convexe. C'est-à-dire que nous pouvons avoir des normalisations inefficaces.

2.2.4 Intersection, union et enveloppe convexe

L'intersection de deux polyèdres convexes est un polyèdre convexe. Un polyèdre convexe est l'intersection de plusieurs demi-espaces. L'intersection de deux polyèdres convexes est donc l'intersection de tous les demi-espaces qui les composent.

L'union de deux polyèdres convexes n'est pas nécessairement un polyèdre convexe. La plus petite sur-approximation de cette union de deux polyèdres convexes est leur enveloppe entière, qui peut être remplacée par leur enveloppe convexe dans Q^n. Cette opération est plus facile à effectuer à partir des systèmes générateurs.

Soient P_1 et P_2 deux polyèdres convexes, de systèmes générateurs :

$$\left\{\begin{array}{l} V_1 = \{v_1^1, \ldots, v_{\alpha_1}^1\} \\ R_1 = \{r_1^1, \ldots, r_{\beta_1}^1\} \\ L_1 = \{l_1^1, \ldots, l_{\delta_1}^1\} \end{array}\right. \; et \left\{\begin{array}{l} V_2 = \{v_1^2, \ldots, v_{\alpha_2}^2\} \\ R_2 = \{r_1^2, \ldots, r_{\beta_2}^2\} \\ L_2 = \{l_1^2, \ldots, l_{\delta_2}^2\} \end{array}\right.$$

Leur enveloppe convexe, notée $convex\_hull(P_1, P_2)$, est définie par :

$$\left\{\begin{array}{l} V = \{v_1^1, \ldots, v_{\alpha_1}^1, v_1^2, \ldots, v_{\alpha_2}^2\} \\ R = \{r_1^1, \ldots, r_{\beta_1}^1, r_1^2, \ldots, r_{\beta_2}^2\} \\ L = \{l_1^1, \ldots, l_{\delta_1}^1, l_1^2, \ldots, l_{\delta_2}^2\} \end{array}\right.$$

Pour tester l'exactitude d'une enveloppe convexe, c'est-à-dire son égalité avec l'union, nous pouvons utiliser la propriété suivante :

Propriété 2.2.1 *Soient P_1 et P_2 deux polyèdres convexes. Soit $P_c = convex\_hull(P_1, P_2)$. Alors $convex\_hull(P_1, P_2) = P_1 \cup P_2$ si et seulement si $P_c \vee \neg P_1 \vee \neg P_2$ n'est pas satisfiable.*

2.3 Ocean, problèmes rencontrés

Il s'agit du programme *ocean.f*, un fichier du benchmark SPEC'95 dont le code source est écrit en Fortran 77 et contient 2997 lignes de code incluant des commentaires, fourni à PIPS comme donnée d'entrée. Les analyses des *transformers* et *préconditions* sont considérées. Cela donne ensuite des calculs de la satisfiabilité (ou un synonyme faisabilité) et des enveloppes convexes des systèmes de contraintes. En manipulant les systèmes de contraintes, nous utilisons en plus la projection et la normalisation.

Dans chaque opération, en cas d'un overflow, c'est-à-dire que les chiffres à traiter sont trop grands, cela entraîne une exception. Avec cette exception, le problème sera indéterminé, ou bien indécidable. Ce genre de problème est déjà traité dans PIPS.

Dans un autre cas où nous avons des explosions avec la projection et/ou le calcul des enveloppes convexes, le problème peut être résolu, mais dans un temps inacceptable. Si l'explosion est trop grave, nous avons une manque de mémoire, qui est considéré comme une faute fatale et le programme s'arrête.

Normalement, en cas d'une explosion, les systèmes de contraintes grandissent, et il y aura peut-être des overflows. Il faut citer tout de suite que les problèmes du temps d'exécution et des overflows s'introduissent dans l'analyse d'*ocean* en même temps.

Nous tentons de traiter les deux cas, pour que le programme ne s'arrête jamais. Il faut par conséquence introduire des time-outs au niveau de l'algorithme.

2.4 Travaux antérieurs - Janus et PIP

La question sur la satisfiabilité d'un système de contraintes est connue depuis très longtemps, et il y a beaucoup d'études sur ce genre de question. Cependant, nous n'avons pas encore une solution optimale dans tous les cas, surtout avec de grands systèmes de contraintes. Il y a des projets qui continuent à chercher une implantation adéquate, en considérant des types de système de contraintes, pour utiliser les méthodes comme Simplexe, Fourier-Motzkin.

Par exemple, Janus [?] développé par Jean-Claude Sogno, est un logiciel qui calcule la satisfiabilité des systèmes de contraintes. Il utilise l'algorithme du Simplexe, mais avec des stratégies de pivotage différentes, et prend en compte le caractère entier des variables.

Nous avons également utilisé le projet PIP [?] développé par Paul Feautrier, dont la libraire est intégré dans PIPS, avec la version de 64 bits, qui est aussi mis en comparaison.

Une autre approche, l'implantation de la méthode du Simplexe en utilisant une définition des Q matrices [?], est mise en considération. Cependant, cette approche a une limite regrettable, c'est qu'on ne peut pas obtenir l'implantation de la méthode pour tester.

Remarque sur la Polylib : Ce stage a commencé avec plusieurs problèmes rencontrés alors qu'on n'avait pas encore décidé la priorité. Dans la recherche bibliographique, nous avons trouvé d'autres solutions pour les problèmes posés dans LINEAR. Dans ce cas, c'est avec le calcul des enveloppes convexes. Il concerne une autre librairie utilisée par LINEAR, qui s'appelle Polylib [?].

Il faut remarquer qu'on est en train de développer la librairie. Au début de ce stage, nous avons utilisé une veille version, et ensuite, nous avons changé pour une version plus récente. Il y a d'autres problèmes concernant la librairie Polylib, qui est utilisé pour le calcul des enveloppes convexes.

En fait, dans les problèmes concernant *ocean*, les calculs des enveloppes convexes prennent le plus du temps. Nous souhaitons avoir une autre librairie qui est plus forte que Polylib, qui fournit les méthodes plus puissantes et efficaces. Nous avons étudier les perspectives du "polka", "qhull", ... Cependant, à cause du temps, nous ne traitons que la satisfiabilité dans ce stage.

Chapitre 3

Méthodologie

3.1 Constitution d'une base de tests

Nous avons mis pris en considération quatre méthodes calculant la satisfiabilité d'un système de contraintes. Ce sont Simplexe et Fourier-Motzkin implémentées dans LINEAR, Simplexe implémentée dans le logiciel Janus, et le logiciel PIP, version 64 bits.

Comme les formats internes des logiciels sont différents, il faut convertir l'un vers l'autre. Cela n'est pas un travail facile, nous avons donc choisi de diviser en deux étapes.

D'abord, il faut construire une base de tests avec un nombre limite des systèmes de contraintes. Après avoir converti les formats des fichiers d'entrée dans cette base, il faut tester la satisfiabilité de ces systèmes de contraintes par ces méthodes. La constitution de la base de tests a pour but de comparer les méthodes selon les critères de pivotage (pour Simplexe), de temps d'exécution, et d'espace. Les résultats seront une source importante pour décider la solution optimale répondant à la question.

Selon la conclusion de ces tests, nous allons choisir la méthode qui nous montre qu'elle est peut-être la meilleure. S'il en existe une, il faudrait ensuite l'intégrer dans LINEAR, pour une comparaison complète. Avec cette comparaison, nous pouvons donc avoir une réponse exacte. Sinon nous devrions chercher un meilleur heuristique en comparaison avec celui qui est actuellement dans LINEAR.

Comment peut-on construire une telle base de tests ? Nous décidons de prendre *ocean* et ses calculs comme une base expérimentale.

Nous avons choisi d'extraire les systèmes de contraintes qui posent des problèmes de satisfiabilité dans *ocean*, et les garder sous forme des fichiers d'entrée dans la base. Il y a vingt-huit systèmes. Cependant, avec *ocean* seul, les systèmes de contraintes ne fabriquent pas tous les cas souhaités. Par exemple, pour les systèmes de contraintes qui sont insatisfiables, ou indéteminés par Simplexe et Fourier-Motzkin dans LINEAR. C'est pourquoi nous avons dû en chercher à partir d'autres sources.

Nous avons ajouté cinq systèmes de contraintes à partir de *ocean*, qui sont insatisfiables (testés par Simplexe ou Fourier-Motzkin dans LINEAR) pour tester par Janus, afin de prouver la cohérence entre ces méthodes. Nous avons également ajouté un système de contraintes qui est indéterminé par LINEAR (y compris Simplexe et Fourier-Motzkin), et qui fait planter Janus, en dehors de *ocean*. Voyons le tableau de la base de tests dans la figure **??**, dans laquel nous notons : LS - Simplexe dans LINEAR ; LS inverse - Simplexe inverse dans LINEAR ; LFM - Fourier-Motzkin dans LINEAR.

Nous allons tester les algorithmes en donnant les fichiers d'entrée d'un même système de contraintes après la conversion. Par conséquence, il faut mettre le temps de la conversion en considération. Chaque test d'une méthode est effectué dans dix itérations pour que le temps d'exécution soit significatif, et répété dix fois. Le temps d'exécution minimale est pris, pour éviter l'influence indésirable. Les tests sont effectués sur une machine UltraSparc IIi 270 MHz 128 MB, utilisant le système d'exploitation SunOS Edition 5.8 Generic_108528-10.

3.2 Instrumentations

3.2.1 Impression d'un système de contraintes

Une structure très importante dans PIPS est Psystem, qui contient la représentation d'un système de contraintes. Nous pouvons imprimer les systèmes de contraintes au cas nécessaire avec les fonctions d'impression. Cependant, il y avait des incohérences entre eux. Les noms des variables imprimés étaient ambigus. Nous sommes arrivés à les afficher avec leurs noms d'origine, ce qui nous a beaucoup aidé à comprendre les systèmes.

Nous avons également mis à jour les routines, avec l'option de l'impression ou non dans un fichier, pour construire une base de tests au format des fichiers d'entrée, qui sont lisibles par des fonctions dans LINEAR.

3.2.2 Exception time-out

Nous avons mis en place des time-outs au niveau des algorithmes qui provoqueront des exceptions, au lieu d'arrêter le programme dans certains cas. Il est possible de changer le temps d'un time-out par une variable d'environnement, pour qu'il n'y ait pas un arrêt imprévu à cause d'une explosion. Ceci dépend malheureusement de la machine qui tourne le programme, de la vitesse et de la mémoire disponible. Le

sc ocean	LS	LS_inverse	Janus	pip64	LFM
67	over flow	faisable	faisable	faisable	faisable
68	over flow	faisable	faisable	faisable	faisable
2388	over flow	faisable	faisable	faisable	faisable
2449	over flow	faisable	faisable	faisable	faisable
2450	over flow	faisable	faisable	faisable	faisable
2458	over flow	faisable	faisable	faisable	faisable
2493	over flow	faisable	faisable	faisable	faisable
2516	faisable	over flow	faisable	faisable	faisable
2524	over flow	faisable	faisable	faisable	faisable
2539	over flow	faisable	faisable	faisable	faisable
2541	over flow	faisable	faisable	faisable	faisable
2585	over flow	faisable	faisable	faisable	faisable
2594	over flow	over flow	faisable	faisable	faisable
2615	over flow	faisable	faisable	faisable	faisable
2945	over flow	faisable	faisable	faisable	faisable
2946	over flow	faisable	faisable	faisable	faisable
5266	over flow	faisable	faisable	faisable	faisable
5327	over flow	faisable	faisable	faisable	faisable
5328	over flow	faisable	faisable	faisable	faisable
5336	over flow	faisable	faisable	faisable	faisable
5371	over flow	faisable	faisable	faisable	faisable
5394	over flow	faisable	faisable	faisable	faisable
5402	over flow	faisable	faisable	faisable	faisable
5417	over flow	faisable	faisable	faisable	faisable
5419	over flow	faisable	faisable	faisable	faisable
5463	over flow	faisable	faisable	faisable	faisable
5472	over flow	over flow	faisable	faisable	faisable
5493	over flow	faisable	faisable	faisable	faisable
3797	infaisable	infaisable	infaisable	infaisable	infaisable
3883	infaisable	infaisable	infaisable	infaisable	infaisable
4809	infaisable	infaisable	infaisable	infaisable	infaisable
4810	infaisable	infaisable	infaisable	infaisable	infaisable
4811	infaisable	infaisable	infaisable	infaisable	infaisable
hydro	overflow	overflow	overflow	infaisable	infaisable
test	overflow	overflow	overflow	infaisable	overflow

FIG. 3.1 – Base de tests

temps du time-out devrait également nous assurer qu'il y ait une explosion avec une certitude la plus grande possible.

Nous avons mis les time-outs dans le calcul de la satisfiabilité, des enveloppes convexes, et dans la projection.

3.2.3 Changement de méthode en cas d'échec

Dans LINEAR, afin de vérifier la satisfiabilité des systèmes de contraintes, après la décision de l'heuristique actuel, nous utilisons soit Simplexe, soit Fourier-Motzkin.

S'il y a un overflow avec Simplexe, nous transférons l'overflow à PIPS. Cela veut dire qu'on n'arrive pas à résoudre le problème avec LINEAR. C'est pareil dans le cas ou Fourier-Motzkin échoue : s'il y a une explosion avec Fourier-Motzkin, nous disons que le système de contraintes n'est pas décidable par LINEAR.

En réalité, après des tests, nous avons trouvé qu'il y a des systèmes de contraintes, qui sont indéterminés par Simplexe, mais décidable par Fourier-Motzkin, et vise versa.

Il y a logiquement une solution temporaire qu'on peut essayer avec autre méthode quand la première a échoué. Nous avons implémenté cette solution dans LINEAR.

3.3 D'autres implantations

3.3.1 Problème de magnitude - un essai avec preprocess

Le problème des overflows apparaît souvent avec Simplexe, surtout avec de grands systèmes de contraintes à manipuler. Une des raisons de ce problème, c'est que les coefficients sont assez grands.

Lorsqu'un système de contraintes est donné au LINEAR, il est projeté le long de certaines variables, pour qu'il soit plus adapté. Mais cela peut produire des coefficients plus grands qu'avant. Preprocess, c'est un petit essai de faire l'inverse la projection, afin de réduire les coefficients qui sont trop grands. Nous pouvons appliquer preprocess sans avoir projeté.

Nous avons essayé d'implémenter preprocess, qui donne seulement quelques résultats modestes. Ces résultats sont montrés par la figure **??**. Il faut encore l'étudier plus profondément.

3.3.2 Problème de l'ordre de contraintes

Dans un projet, le problème de la compatibilité est très important, surtout dans les grands projets. Nous avons détecté que les fonctions de duplication des structures

sc	LS	LS_inverse	pre_process	pre_process
ocean			MAX coeff = 113	MAX coeff = 131
67	over flow	faisable	over flow	over flow
68	over flow	faisable	over flow	over flow
2388	over flow	faisable	faisable	faisable
2447	faisable	faisable	over flow	over flow
2449	over flow	faisable	faisable	faisable
2450	over flow	faisable	faisable	faisable
2458	over flow	faisable	over flow	over flow
2493	over flow	faisable	faisable	faisable
2516	faisable	over flow	faisable	faisable
2524	faisable	faisable	over flow	over flow
2539	over flow	faisable	over flow	over flow
2541	over flow	faisable	over flow	over flow
2585	over flow	faisable	over flow	over flow
2594	over flow	over flow	over flow	over flow
2615	over flow	faisable	over flow	over flow
2945	over flow	faisable	over flow	over flow
2946	over flow	faisable	over flow	over flow
5266	over flow	faisable	faisable	faisable
5272	faisable	faisable	over flow	faisable
5273	faisable	faisable	over flow	faisable
5325	faisable	faisable	over flow	over flow
5327	over flow	faisable	faisable	faisable
5328	over flow	faisable	faisable	faisable
5336	over flow	faisable	over flow	over flow
5371	over flow	faisable	faisable	faisable
5394	over flow	faisable	faisable	faisable
5402	over flow	faisable	over flow	over flow
5417	over flow	faisable	over flow	over flow
5419	over flow	faisable	over flow	over flow
5463	over flow	faisable	over flow	over flow
5472	over flow	over flow	over flow	over flow
5493	over flow	faisable	over flow	over flow
2394	faisable	faisable	over flow	faisable
2395	faisable	faisable	over flow	faisable

FIG. 3.2 – Résultats du preprocess

de base ont renversé l'ordre de chaque élément. En fait, ces fonctions n'ont pas vraiment fabriqué les mêmes objets. Ici, on a payé pour la clarté et simplification du code, en utilisant les listes dans le langage de programmation C.

Nous verrons plus loin que la méthode du Simplexe est sensible à l'ordre des composantes d'un système de contraintes. Mais nous ne pouvons pas toutes remplacer ces anciennes versions, pour une raison de compatibilité. Cependant, nous avons installé la version corrigée de ces fonctions et les avons utilisées dans plusieurs cas. Il faut mentionner ici que l'implantation de la méthode Simplexe dans LINEAR n'est pas tout à fait correcte. En fait, il y a de petites fautes que nous avons corrigées, mais nous ne pouvons pas assurer qu'il n'y en a plus.

3.3.3 Convertisseurs

Pour tester les systèmes de contraintes dans LINEAR, nous les avons imprimés et stockés sous forme de fichiers d'entrée dans le format de LINEAR. Nous devons avoir des tests en dehors de LINEAR, qui lisent ces fichiers, les convertissent au format interne dans LINEAR afin de tester la satisfiabilité. Ensuite, nous allons tester ces systèmes de contraintes par Janus et PIP64 avec leurs formats différents.

Il nous faut donc des convertisseurs qui, à partir d'un système de contraintes au format dans LINEAR, créent les fichiers d'entrée au format du Janus et du PIP64, en assurant que les données d'entrée sont équivalentes. Il faut ensuite des fonctions dans LINEAR, pour convertir les formats internes, pour qu'on puisse appeler directement dans LINEAR, ou bien dans des tests. Enfin, pour les intégrer dans LINEAR, il faut traiter tous les cas comme exception.

Chapitre 4

Satisfiabilité

4.1 Satisfiabilité dans LINEAR

Il existe plusieurs algorithmes pour effectuer les tests de faisabilité. Dans LINEAR, nous utilisons soit la méthode Simplexe lorsque le nombre de contraintes du système devient trop important, ou lorsqu'elles sont très liées, soit la méthode d'élimination de Fourier-Motzkin. Ce sont des tests approchés.

4.1.1 Méthode du Simplexe

Pour comprendre la méthode du Simplexe, nous donnons quelques définitions comme suit :

Définition 4.1.1 *Si A est une matrice de dimension $m \times n$, b un vecteur à m composantes et c un vecteur à n composantes, alors la recherche d'un vecteur $x = (x_1, \ldots, x_n)$ satisfaisant $Ax \leq b$ en rendant maximum la fonction $z = cx$ est un problème de programmation linéaire. Un tel problème se note de la manière suivante :*

$$\{\max z = cx \mid Ax \leq b\}$$

Les *solutions admissibles* sont celles qui satisfont toutes les contraintes du problème. Elles se distinguent des *solutions optimales* qui, parmi les solutions admissibles, optimisent la fonction dont on cherche le maximum ou le minimum. La fonction linéaire que l'on optimise est appelée *fonction objectif* ou *fonction économique*.

Un problème est dit *linéaire* si les deux conditions suivantes sont satisfaites :

1. f(X) est une fonction linéaire des variables où les coordonnées sont les données du problème.
2. Les contraintes sont linéaires par rapport aux variables. Elles sont donc de la forme $\sum a_{ij}x_j \leq b_i$

L'algorithme de Simplexe est utilisé mondialement et amélioré de diverses manières. Il permet de trouver, s'il existe, l'ensemble des valeurs pour les variables qui satisfont toutes les contraintes d'un système de grande taille en des temps de calcul relativement faibles.

Comme nous avons mentionné, un système de contraintes est en fait un polyèdre convexe avec ses sommets, ses rayons et ses droites. Un système de contraintes est satisfiable si et seulement si le polyèdre correspondant n'est pas vide. L'idée de la méthode du Simplexe est de traverser tous les sommets du polyèdre le long des rayons ou droites, jusqu'à ce qu'une solution optimale soit trouvée. À chaque pas, on doit vérifier si la solution est optimale, sinon on pivote un autre sommet selon un critère quelconque.

Dans LINEAR, nous ne nous intéressons pas encore à trouver la solution optimale, mais seulement la satisfiabilité. Cependant, il faut trouver une solution initiale avant de pivoter. Si on trouve une solution initiale, cela veut dire que le système de contraintes est satisfiable. Qu'est-ce qu'on doit faire ?

Etant donné le problème de programmation linéaire, nous choisissons la forme équivalente :

$$\max\{cx \mid x \geq 0 \; Ax \leq b\}$$

Nous pouvons écrire $Ax \leq b$ comme $A_1x \leq b_1$, $A_2x \geq b_2$ avec $b_1 \geq 0$ et $b_2 < 0$.

Considérons ce nouveau problème :

$$\max\{\mathbf{1}(A_2x - \tilde{x}) \mid x, \tilde{x} \geq 0 \; A_2x - \tilde{x} \leq b_2\}$$

où $\tilde{\mathrm{x}}$ est un vecteur représentant de nouvelles variables, et $\mathbf{1}$ est un vecteur unitaire. Alors l'ensemble $\mathrm{x} = 0$ et $\tilde{\mathrm{x}} = 0$ définit une solution du nouveau problème. On peut ensuite utiliser la méthode du Simplexe avec cette solution initiale. Si la valeur maximale pour le nouveau problème est $\mathbf{1}b_2$, nous avons tout de suite la solution correspondante pour le premier problème.

La méthode du Simplexe présente plusieurs variantes dont la méthode algébrique et la méthode par tableaux. La méthode par tableaux est utilisée dans LINEAR. Elle débute par la formation d'un tableau. L'hypothèse de départ est la suivante : on part d'une base et d'une solution de base réalisable avec un tableau de départ associé. Par convention la ligne 0 du tableau correspond à la ligne de la fonction économique. On pivote en changeant le tableau, jusqu'à ce que la condition quelconque soit satisfaite.

Comme les systèmes de contraintes s'étendent de proche en proche la frontière où nous avons facilement des overflows, nous nous sommes aperçu que la méthode Simplexe est sensible à l'ordre de contraintes. Ici, il faut rappeler qu'une version des fonctions de duplication des structures de base est mal construite, y compris la duplication d'un système de contraintes - sc_dup et sc_copy.

4.1.2 Méthode de Fourier-Motzkin

Etant donné que l'élimination de Fourier-Motzkin marche bien avec les égalités, il est naturel de chercher une méthode similaire qui travaille avec des inégalités. Cette méthode a été trouvé par Fourier[1827], Dines [1918] et Motzkin[1936]. Elle est parfaitement adaptée pour résoudre le problème comme : Soit une matrice A et un vecteur b. Tester si $Ax \leq b$ a une solution. S'il en existe, exhiber une.

Soit A une matrice de dimension $m \times n$, b un vecteur à m composantes, comme nous pouvons multiplier chaque inégalité par un nombre scalaire positif, nous pouvons assumer que tous les éléments dans la première colonne dans A sont 0 ou ± 1. Le problème équivalent :

$$\begin{cases} \xi_1 + a_i x' \leq \beta_i, (i = 1, \ldots, m') \\ -\xi_1 + a_i x' \leq \beta_i, (i = m' + 1, \ldots, m'') \\ a_i x' \leq \beta_i, (i = m' + 1, \ldots, m'') \end{cases}$$

où x $= (\xi_1, \ldots, \xi_n)^T$, $x' = (\xi_2, \ldots, \xi_n)^T$ et $(a_1, \ldots, a_m)$ sont les lignes dans A avec le premier élément supprimé. Nous avons ensuite :

$$\max(a_j x' - \beta_j) \leq \xi_1 \leq \min(\beta_i - a_i x') \quad (m' + 1 \leq j \leq m'', 1 \leq i \leq m')$$

Comme résultat, nous pouvons éliminer la variable inconnue ξ_1 dans le système. Continuer cette procédure jusqu'à ce que nous avons seulement une variable, nous avons la réponse triviale. Cette méthode n'est pas polynomiale mais exponentielle dans le cas général.

4.2 Problèmes

4.2.1 Heuristique entre Simplexe et Fourier-Motzkin dans LINEAR

Comme nous avons déjà mentionné, il y a deux méthodes calculant la satisfiabilité d'un système de contraintes dans LINEAR : Fourier-Motzkin et Simplexe. Fourier-Motzkin est adéquat pour les égalités, qui sont seulement projetées, mais peut exploser avec les inégalités quand on les combine. Cependant, cette méthode est assez rapide pour traiter des systèmes avec quelques inégalités.

Simplexe convertit toutes les égalités en inégalités et ajoute des hyperplans. Par conséquence, cette méthode n'est pas approprié avec les systèmes de contraintes contenant beaucoup d'égalités. Même si on peut toujours le faire, on a tendance à projeter les égalités autant que possible. D'autre part, cette méthode est un peu lente avec de petits systèmes de contraintes, à cause de la matrice dense qu'on doit créer et manipuler.

Dans l'heuristique actuelle, nous comptons le nombre d'égalités et le nombre d'inégalités. Si le nombre d'inégalités est plus grand que dix, nous allons choisir Simplexe, à condition que le nombre d'égalités est moins que six. Sinon, nous allons choisir Fourier-Motzkin. Dans le cas où le nombre d'inégalités est plus grand que dix et le nombre d'égalités est plus grand que six, nous allons projeter les égalités autant que possible, puis utiliser Simplexe.

Nous nous apercevons que l'heuristique ne prend en compte pas ni la dimension, ni le poids de chaque système de contraintes qui sont aussi importants. Nous devrions remettre en question les problèmes de magnitude des coefficients, surtout pour la complexité en temps et en espace mémoire du calcul. L'heuristique devrait compter la grande possibilité d'un overflow quand les systèmes de contraintes à traiter sont grands.

4.2.2 Vitesse et espace mémoire

Dans un logiciel assez grand comme PIPS, les problèmes de vitesse et d'espace mémoire utilisée sont très importants. En particulier, le calcul de la satisfiabilité est une opération de base, qui est appelée des milliers des fois dans une session de travail dans PIPS. Il faut absolument que cette fonction soit la plus rapide possible.

Ici, nous utilisons l'allocation dynamique, pour économiser de la mémoire. Cela entraîne pourtant une vitesse plus modeste, en comparant avec l'utilisation de l'allocation de mémoire statique.

4.2.3 Overflow et exception

Il existe un facteur important que les systèmes de contraintes à manipuler dans PIPS sont assez grands, même avec d'autres logiciels comme Janus et PIP. Des cas dans les calculs de la satisfiabilité d'un système de contraintes, les chiffres doivent être multipliés, et si leur produit dépasse la limite (64 bits), nous avons un overflow. En plus, les chiffres peuvent être accumulés et deviennent trop grands, si bien qu'on ne peut plus les traiter.

Le cas ou le système de contraintes donné au LINEAR génère un overflow, il nous faut une exception pour dire que le programme ne peut pas résoudre le problème. Le résultat est donc indéterminé.

4.3 Comparaisons avec Janus et PIP64

4.3.1 Critère de pivotage

Dans Janus, Jean-Claude Sogno a intégré l'élimination des égalités et des inégalités dans le calcul de la satisfiabilité de façon très efficace. Après ces éliminations, le système de contraintes donné au pivotage, le calcul sous forme de tableaux dans l'algorithme du Simplexe devient assez petit. Par contre, l'implantation de Simplexe dans LINEAR travaille toujours sur de grands systèmes de contraintes. Le nombre de pivotages est alors assez grand.

Un point remarquable dans Janus est que nous pouvons choisir la stratégie d'élimination et de pivotage.

1. Élimination d'égalités (toutes les égalités sont éliminées) (*)
2. Élimination d'inégalités
   (a) Pas d'élimination
   (b) Cas trivial
   (c) Nombre d'inégalités décroît (*)
   (d) Nombre d'inégalités n'augmente pas
3. Pivotage (prime et dual*)
   (a) stratégie Simplexe (plus petit pivot ou plus grand pivot*)
   (b) Pivot unitaire, sinon stratégie Simplexe
   (c) Pivot unitaire, sinon variables ajoutées

Nous avons choisi les options marquées par * pour tester nos systèmes de contraintes. Les nombres de pivotage du Simplexe dans LINEAR, avec les nombres d'itérations contenant les nombres d'éliminations et de pivotages que Janus effectue, sur les vingt-six systèmes de contraintes dans la base de tests sont comparés dans Figure **??**.

Si nous prenons une itération d'élimination, ou de pivotage comme une opération de base, nous voyons que Janus est le plus efficace. Pour PIP64, nous n'avons pas encore étudié.

4.3.2 Critère de temps

Le critère le plus important dans ces comparaisons, c'est le critère de temps d'exécution, puisque nous souhaitons avoir les méthodes les plus efficaces possibles. Nous prenons des fichiers d'entrée de formats différents, mais ils représentent les

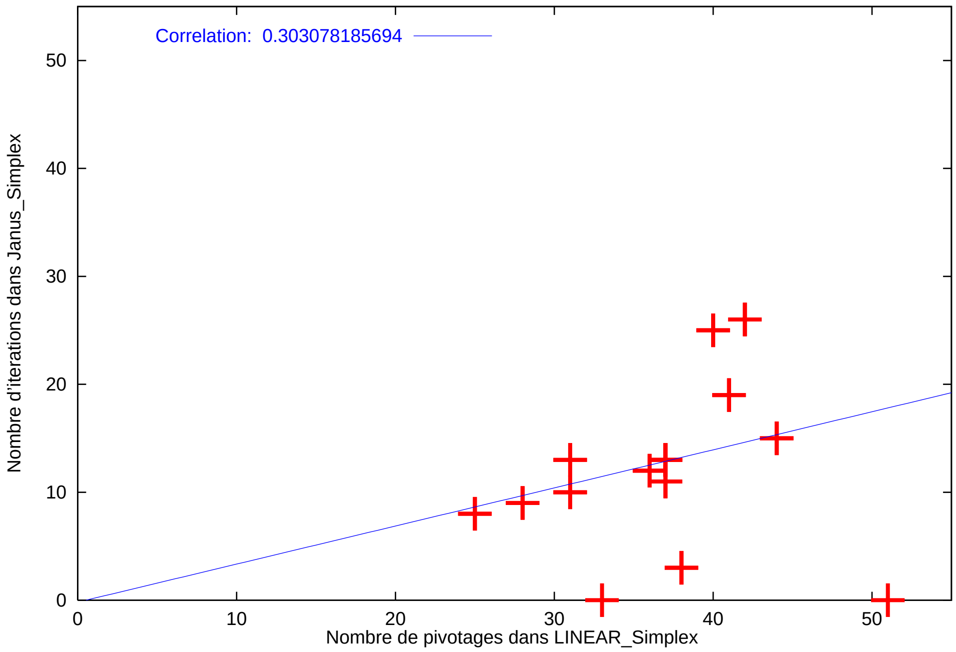

FIG. 4.1 – Nombre de pivotage du Simplexe dans LINEAR et nombre d'itérations dans Janus

mêmes systèmes de contraintes. En fait, la comparaison est théoriquement juste. Cependant, le but de ces comparaison, c'est que nous allons implémenter les méthodes dans LINEAR, c'est pourquoi il nous faut également compter le temps de conversion de format.

La figure **??** représente la comparaison entre Simplexe dans LINEAR avec Simplexe dans Janus. Il n'y a aucune corrélation entre les deux implantations du Simplexe, parce qu'en fait, Janus utilise une projection plus efficace que LINEAR. Il projette les variables, les contraintes autant que possible, quant à LINEAR, nous les projetons de façon très simple. De plus, nous voulons remarquer que même si nous avons ajouté le temps de conversion de format, qui n'est normalement pas très varié par rapport au grandeur du système de contraintes, Janus est toujours plus stable et plus efficace.

Maintenant, nous allons comparer les deux méthodes qui sont déjà implémenté dans LINEAR. Il faut aussi dire que nous avons essayé d'utiliser seulement Fourier-Motzkin dans les calculs du fichier d'entrée *ocean*, et il n'y a aucune exception. Cela veut dire que Fourier-Motzkin peut résoudre tous les problèmes de satisfiabilité dans *ocean*. Mais cela ne veut pas dire que Fourier-Motzkin n'échoue pas dans d'autres cas. Au contraire, comme déjà mentionné dans la section de l'heuristique, il y a autant de systèmes de contraintes qui ne sont pas du tout appropriés pour Fourier-Motzkin.

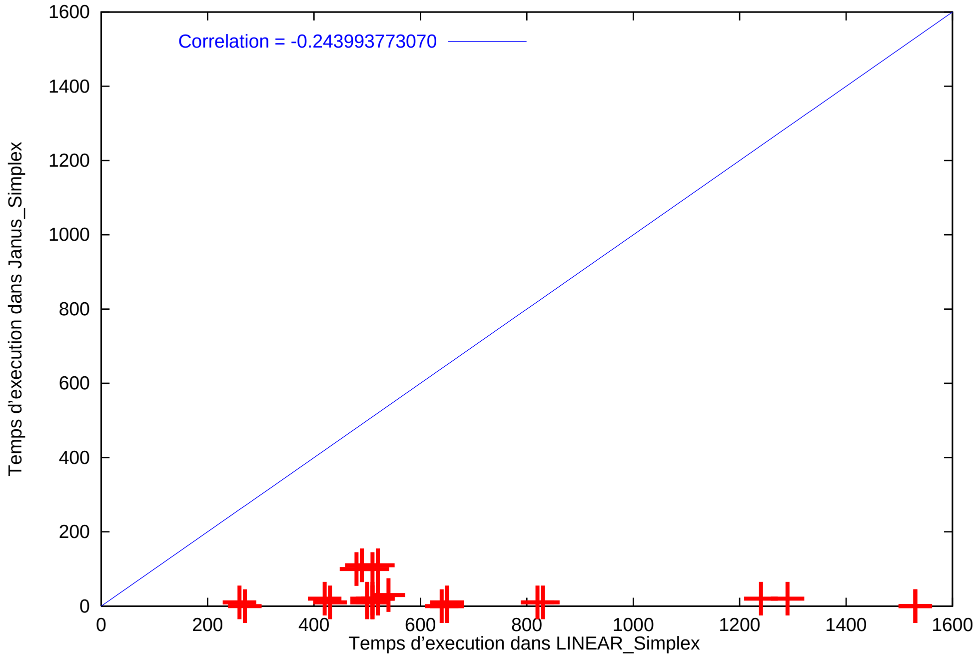

FIG. 4.2 – Simplexe dans LINEAR et Simplexe dans Janus

C'est-à-dire qu'il nous faut chercher une autre solution.

La figure ?? représente la comparaison entre Simplexe dans LINEAR et Fourier-Motzkin dans LINEAR. Il est évident que Fourier-Motzkin est plus efficace que Simplexe, sauf pour quatre systèmes de contraintes. Aucun remarque sur le genre de ces systèmes sauf qu'ils ne soient pas bon pour la projection. Ceci est aussi vrai pour Janus, parce qu'il a passé le plus de temps pour calculer ces systèmes.

La figure ?? représente la comparaison entre Fourier-Motzkin dans LINEAR et Simplexe dans Janus. Il y a une corrélation très remarquable entre ces deux implantations. A part le temps de la conversion du format interne, on peut voir quand même que Simplexe dans Janus est plus efficace que Fourier-Motzkin dans LINEAR. C'est plus logique, en comparaison avec Simplexe dans LINEAR, surtout quand nous savons que Simplexe est la méthode la plus utilisée dans les calcul de satisfiabilité avec de grands systèmes de contraintes.

Ensuite, nous avons d'autres figures : la figure ?? représente la comparaison entre Fourier-Motzkin dans LINEAR et Simplexe dans PIP64 ; la figure ?? représente la comparaison entre Simplexe dans LINEAR et Simplexe dans PIP64 ; la figure ?? représente la comparaison entre Simplexe dans LINEAR et Simplexe dans PIP64.

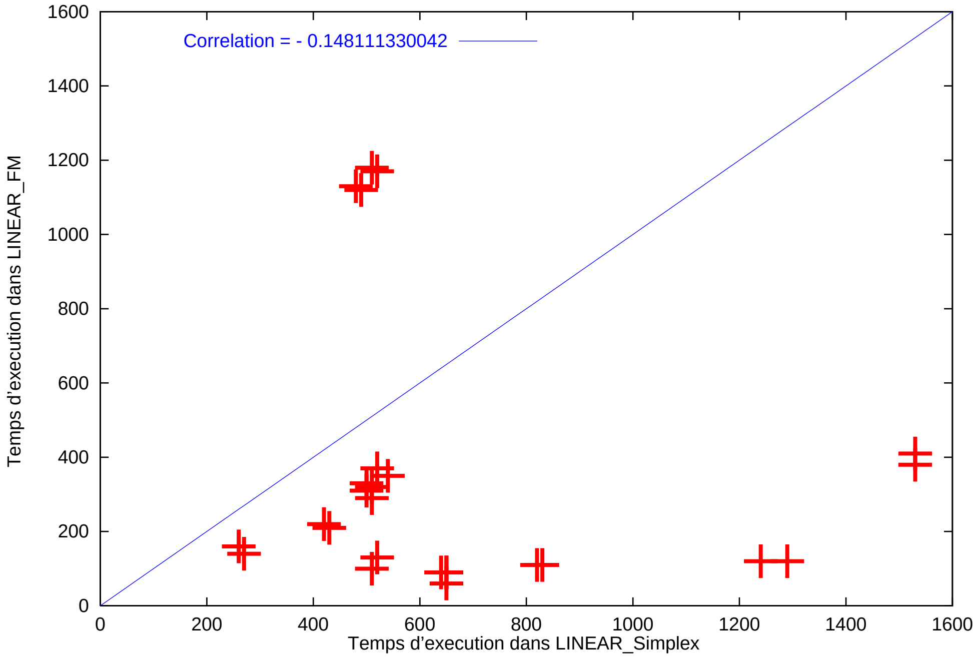

FIG. 4.3 – Simplexe dans LINEAR et Fourier-Motzkin dans LINEAR

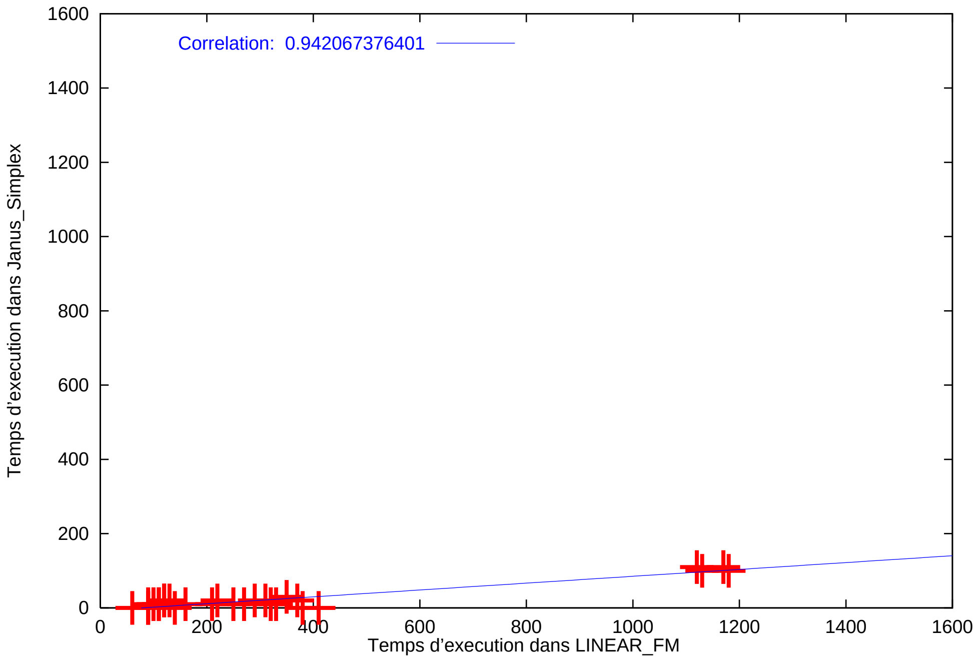

FIG. 4.4 – Fourier-Motzkin dans LINEAR et Simplexe dans Janus

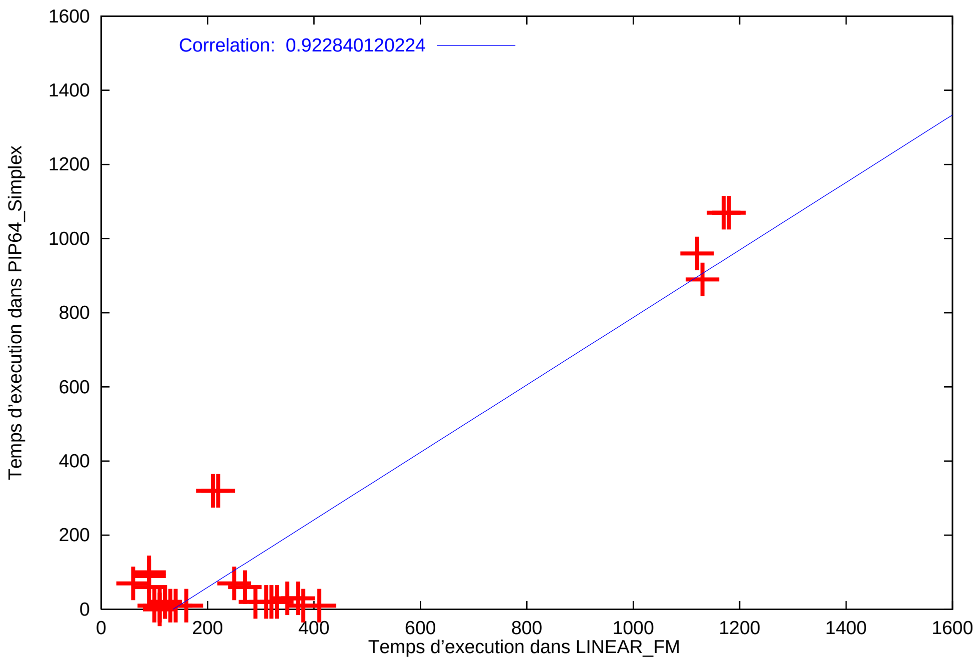

FIG. 4.5 – Fourier-Motzkin dans LINEAR et Simplexe dans PIP64

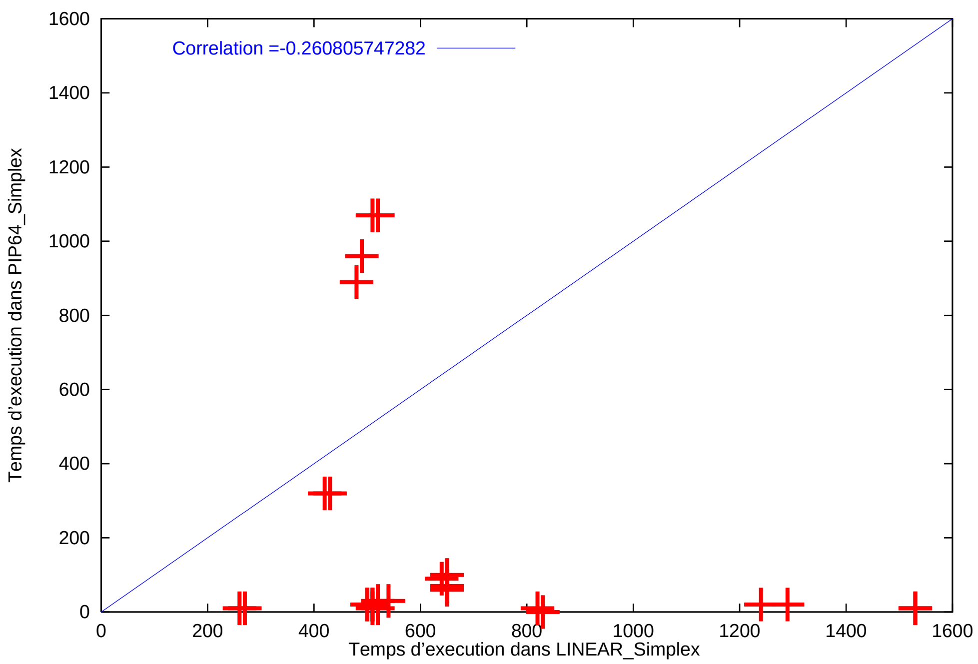

FIG. 4.6 – Simplexe dans LINEAR et Simplexe dans PIP64

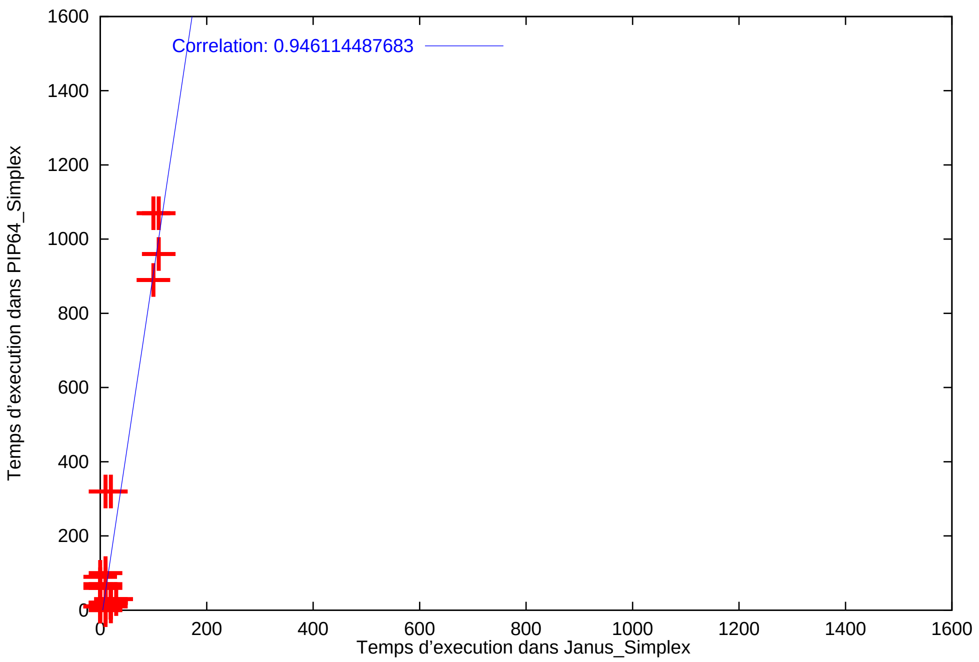

FIG. 4.7 – Simplexe dans Janus et Simplexe dans PIP64

4.3.3 Critère d'espace mémoire

Considérons la structure de donnée dans Janus, nous voyons que Janus utilise des tableaux avec deux dimensions fixées par AKCOLONNES et AKLIGNES qui sont définis comme suit : AKCOLONNES = MAXCOLONNES + 10 ; AKLIGNES = MAXLIGNES + 10, où nous avons mis MAXCOLONNES = 108, MAXLIGNES = 250.

Dans Janus, toutes les informations comme le système de contraintes, la stratégie de pivotage, la solution (si nous en trouvons une) sont stockées dans une structure qui s'appelle *iprobleme*. Nous avons ensuite les comparaisons suivantes entre LINEAR et Janus :

1. Simplexe et Fourier-Motzkin dans LINEAR
   (a) Allocation de mémoire dynamique par l'utilisation de *malloc()*.
   (b) Libération manuelle.
   (c) Pour chaque système de contraintes, nous avons un espace de mémoire flexible pour tous système de contraintes.
2. Simplexe dans Janus
   (a) Allocation de mémoire statique par l'utilisation de *struct*.
   (b) Libération automatique.

(c) Pour tous les systèmes de contraintes, nous allouons une même espace de mémoire.

Nous voyons facilement que l'implantation de Janus au point de vue d'espace mémoire n'est pas vraiment bonne. Nous allons perdre beaucoup d'espace mémoire afin de traiter tous les systèmes de contraintes. En plus, au cas où un système à traiter est trop grand (souvent dans PIPS), il dépasse la limite des tableaux. Il nous faut donc des exceptions.

Finalement, c'est un problème de l'implantation, on peut l'améliorer dans l'avenir.

4.4 Intégration de Janus dans PIPS

Jusqu'à maintenant, nous n'avons que testé la validation de l'intégration, comparé le temps d'exécution avec quelques petits exemples. Il faut du temps pour les grands exemples, qui s'exécutent pendant quelques jours.

Nous voudrions tester trois versions de LINEAR : l'ancienne, celle avec le changement de méthode au cas d'un échec et celle avec l'utilisation du Janus. Mais le problème, c'est que la réponse de satisfiabilité d'une phase peut entraîner un chemin d'exécution, alors le temps d'exécution, tout à fait différent. L'ancienne version de LINEAR donne une réponse qui est peut-être fausse lorsque nous avons un overflow.

C'est pourquoi nous ne considérons que les deux dernières. Nous avons testé trois fois et pris le temps d'exécution minimum pour les deux benchmarks : *ocean* and *tomcatv*, le dernière est venu du SPEC95 CFP benchmarks.

1. Les overflows avec *ocean* :
   (a) Sans Janus : 28 systèmes de contraintes.
   (b) Avec Janus : 26 systèmes de contraintes
2. Temps d'exécution avec *tomcatv* :
   (a) Changement de méthode : 0m12.650s
   (b) Utilisation de Janus : 0m12.260s

Nous voyons que Janus est un peu plus rapide, même si le temps de conversion de format est pris en compte. Actuellement, nous avons l'intégration du Janus dans la version de développement du LINEAR.

Chapitre 5

Conclusions

A partir des problèmes rencontrés, nous avons étudié l'environnement contenant des opérations de base, pour comprendre l'aspect théorique des problèmes. Ensuite, nous avons construit une base de tests afin de mesurer l'efficacité des algorithmes. Grâce à cette base de tests, nous avons pu comparer les quatre algorithmes de satisfiabilité, ce qui nous a conduit à l'intégration dans LINEAR des meilleures tests de satisfiabilités, après les résultats expérimentaux.

Nous avons également fourni quelques outils pour la détection des problèmes avant qu'ils ne surviennent. Nous avons offert une possibilité de réponse plus grande dans la libraire LINEAR, avec le changement de méthode, au cas d'un échec.

Avec ces modifications, le logiciel PIPS est devenu plus consistant dans les calculs de satisfiabilité. Cependant, il nous reste encore d'autres problème à traiter, comme le problème de magnitude, le raffinement de l'intégration du Janus dans LINEAR, etc. En plus, nous devrions aussi améliorer d'autres algorithmes dans la programmation linéaire, comme les calculs des enveloppes convexes, la projection, la normalisation.

Bibliographie

[1] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, and R. Keryell. Get started. pips user manual interface wpips, epips, tpips, jpips. frequently asked questions. Technical report, Centre de Recherche en Informatique, Ecole des Mines de Paris, September 1999. tech.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991. inproceeding.

[3] D. O. Azulay and J. F. Pique. A revised simplex method with integer Q-matrices. *ACM Transactions on Mathematical Software*, 27(3) :350–360, September 2001. article.

[4] F. C. Corinne Ancourt and F. Irigoin. Linear algebra as a proof engine. Technical report, Centre de Recherche en Informatique, Ecole des Mines de Paris, November 2000. tech.

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[6] K. Dowd and C. Severance. *High Performance Computing*. O'Reilly, Sebastopol, 1998. book.

[7] A. et D.H. Marwick. *Programming in Standard Fortran 77*. HEINEMANN EDUCATIONAL, London, 1979. book.

[8] P. Feautrier. Parametric integer programming. project PIP.

[9] K. Fukuda, T. M. Liebling, and C. Lutolf. Extended convex hull. *Computational Geometry*, 20(1-2) :13–23, 2001. article.

[10] A. B. G.J.Badros and P. Stuckey. The cassowary linear arithmetic constraint-solving algorithm. *ACM Transactions on Mathematical Software*, 8(4) :267–306, December 2001. article.

[11] P. group. Logiciel pips. Web site : www.cri.ensmp.fr/ pips/. project PIPS.

[12] P. group. Pips development environment. Technical report, Centre de Recherche en Informatique, Ecole des Mines de Paris, May 2001. tech.

[13] D. K. W. Hervé Le Verge, Vincent Van Dongen. Loop nest synthesis using the polyhedral library. Technical report, Institut National de Recherche en Informatique et en Automatique, May 1994. tech.

[14] V. Loechner. Polylib. Web site : icps.u-strasbg.fr/polylib/. project.

[15] M. Minoux. *Programmation mathématique, théorie et algorithmes, tome 1 et 2*. BORDAS et C.N.E.T.-E.N.S.T., Paris, 1983. book.

[16] G. Rote. Degenerate convex hulls in high dimensions without extra storage. In *ACM Symposium on Computational Geometry*, pages 26–32, June 1992. inproceeding.

[17] A. Schrijver. *Theory of linear and integer Programming*. Wiley-Interscience, New York, 1986. book.

[18] J.-C. Sogno. Logiciel janus. restricted. project Chloé.

[19] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms : why the simplex algorithm usually takes polynomial time. In *ACM SIGACT Symposium on Theory of Computing*, pages 296–305, July 2001. inproceeding.

[20] D. K. Wilde. A library for doing polyhedral operations. Technical report, Brigham Young University, August 1997. tech.