Projet Logique : Equipe COOP

Dhimoila Grégoire et Guyot Jérôme

May 9, 2023

Abstract

L'objectif de ce projet est de déterminer la satisfiabilité de formule basé sur des conditions arithmétiques. On cherchera ainsi par exemple à déterminer s'il existe une valuation rendant la formule suivante vraie : $((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$. Pour ce faire on met en place une coopération bidirectionnelle entre le SAT solveur Glucose, et le solveur d'arithmétique linéaire GLPK.

1 Introduction

La recherche de solutions satisfaisantes pour des problèmes de logique propositionnelle a été au cœur de nombreuses recherches en informatique théorique. Le problème SAT (Satisfiabilité Booléenne) consiste à déterminer s'il existe une assignation de valeurs de vérité aux variables d'une formule logique qui rend cette formule vraie.

Cependant, de nombreux problèmes pratiques nécessitent la manipulation de variables avec des contraintes plus complexes que de simples valeurs booléennes. C'est là que le problème SMT (Satisfiabilité Modulo Théorie) devient important. Le problème SMT vise à déterminer si une formule logique contenant des contraintes de diverses théories peut être satisfaite ou non.

Dans ce projet, nous nous concentrons sur le problème SMT dans le contexte de la théorie arithmétique. Nous cherchons à déterminer la satisfiabilité de formules basées sur des conditions arithmétiques. Pour ce faire, nous utilisons une coopération bidirectionnelle entre le solveur SAT Glucose et le solveur d'arithmétique linéaire GLPK. Notre objectif est de trouver une assignation de valeurs de vérité pour les variables de la formule qui satisfait toutes les contraintes arithmétiques données.

Dans la suite de ce rapport, nous allons décrire les techniques utilisées pour résoudre les problèmes de satisfiabilité dans la théorie arithmétique, en détaillant la coopération entre les solveurs SAT et d'arithmétique linéaire. On appellera SAT-formule les formules avec des variables booléennes et SMT-formules celle avec des variables entières.

2 Description de l'algorithme

On dispose de deux outils : le SAT-solveur glucose, et le solveur d'arithmétique linéaire GLPK. Pour savoir si une formule est satifiable on peut appliquer l'algorithme 1 ci-dessous :

L'algorithme 1 est relativement naïf, il transforme la SMT-formule en un couple SAT-formule,hashtable. Puis il demande à glucose de générer une valuation satisfaisant la SAT-formule tant qu'il y en a, regarde si elle est cohérente arithmétiquement grâce à GLPK, et si elle ne l'est pas, on rajoute une clause dans la sat-formule correspondant à la negation de cette valuation.

Le programme termine bien car il existe un nombre fini de valuation satisfaisant la formule, on va donc bien terminer en renvoyant satisfibiable ou non satisfiable. Pour la correction, si on renvoie une valuation alors elle satisfait la SAT-formule et les contraintes sont cohérentes aritmétiquement, elle fonctionne donc.

Algorithme 1 : Est_satisfiable

Entrées : Un fichier contenant une formule
Sorties : Satisfiabilité de la formule et valuation si satisfiable
SMT-formule ← lecture fichier
(SAT-formule, Hashtable) ← process SMT-formule
(status,model) ← run_glucose SAT-formule
tant que *status* ≠ *Unsatisfiable* **faire**
  problème ← initialisation problème model Hashtable
  (cohérence,valuation) ← run_glpk problème
  si *cohérence = True* **alors**
    return Satisfiable, valuation
  sinon
    new_clause ← negate model // pour dire que cette valuation est mauvaise
    SAT-formule ← add_clause SAT-formule new_clause
    (status,model) ← run_glucose SAT-formule
  fin
fin
return Non Satisfiable

3 Gestion des formules

3.1 Transformation de la SMT-Formule

Une première contrainte est que Glucose prend en entrée des formules au format dimacs et contenant des variables booléennes. Il faut donc transformer les formules initiales de sorte à avoir des formules aux formats dimacs. L'idée de la transformation ici est de passer d'une SMT-formule, vers un couple SAT-formule, hashtable telle que sur la clé x_i on obtient une contrainte arithmétique, où x_i est une variable booléennes de la SAT-formule.

Ainsi la formule $((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$ sera transformée en
$(x1 \vee x2) \wedge x3 \wedge (x4 \vee x5)$, $x1 \rightarrow (a = 1)$; $x2 \rightarrow (a = 2)$; $x3 \rightarrow (a \geq 3)$; $x4 \rightarrow (b \leq 2)$; $x5 \rightarrow (b \geq 3)$

Une fois que l'on a une SAT-formule, l'appel de glucose est très simple, on utilise -model de sorte à ce qu'il renvoie une valuation satisfaisant la formule si possible. Puis on trie ce que renvoie glucose pour juste garder le statut : l'aspect satisfiable de la formule, et le model : la valuation si satisfiable.

3.2 Ajout de clause

Si jamais GLPK renvoie que la valuation proposée est invalide, il faut modifier la SAT-formule pour ne pas renvoyer la même valuation. En notant $v_1v_2...v_n$ la valuation où $v_i = \pm x_i$, on ajoute la clause $(-v_1 \vee -v_2 \vee ... \vee -v_n)$ obligeant glucose à trouver une autre valuation. Cette nouvelle clause est naïve, et grâce aux algorithme de 5.1 on peut trouver un sous ensemble incompatible qui permet alors d'ajouter une meilleure clause. on trouve donc un sous ensemble S de $v_1v_2...v_n$ et on va ajouter la négation des contraintes de S plutôt que celles la valuation.

Pour ce faire on utilise donc une fonction negate, qui va renvoyer la negation d'une contrainte arithmétique. A noter que comme le parseur de GLPK ne connaît pas $\neq$, la negation de $\neq$ correspond à deux contraintes avec $<$ et $>$, car être différent c'est être soit stritement plus petit soit strictement plus grand.

4 Gestion des Lp_problèmes

4.1 Parsing

Pour transformer un littéral en contrainte Lp, on parse de string vers Aexp, puis d'Aexp vers contrainte Lp. Le parser est assez classique, il extrait les blocks gauches et droits de l'opérateur principal,

et construit ainsi récursivement un arbre d'opérations arithmétiques.

Cependant, les problèmes pris en charge par GLPK ne prennent pas de contrainte trop élaborées (avec des parenthèses, typiquement, ou même simplement des $\neq$), nous avons donc opté dans un premier temps pour l'abandon de ce parser pour simplement écrire les contraintes dans un fichier au format d'un problème lp directement. Ainsi, on se limite aux contrainte que comprend le parseur de Lp. Cela réduit drastiquement les possibilités de contraintes, mais dans un premier temps, c'est suffisant dans la mesure ou cela revient à traiter le sujet original. On s'interessera donc aux expressions arithmétiques générales en extension (voir section 5.3).

4.2 Création des problèmes LP

Pour créer les problèmes, une première étape est de créer les contraintes au format Lp, pour cela on se base sur l'algorithme de parsing. Dans la première version, le parser de Lp transforme directement en contrainte Lp, dans la seconde, une fonction arbre_2_contrainte s'occupe de cette transformation. On obtient donc à la fin une liste de contraintes $c\_1, ..., c\_n$.

Une fois la liste des contraintes à disposition on peut créer le problème Lp. Pour cela on doit définir l'objectif et la liste des contraintes. On choisit des problèmes de la forme "maximise 0, subject to $c\_1, ..., c\_n$ ", qui garantissent l'existence d'une solution si et seulement si les contraintes sont compatibles.

Il ne reste plus qu'a appliquer GLPK_solve pour savoir si les contraintes sont compatibles et une valuation compatible si elle existe.

4.3 Erreurs de GLPK

Certaines exécutions de GLPK_solve donnent des résulats surprenants.

Par exemple :

Formule : (a $<$ 7 or b $>$ 7) and (a $>$ 7 or b $<$ 6) and (a $<$ 6 or b $<$ 6) and (a $<$ 6 or b $>$ 7)

Contenu de problem.lp :
maximize 0 ; subject to a$<$7 ; b$\leq$7 ; a$\leq$7 ; b$<$6 ; a$\geq$6 ; b$<$6 ; a$<$6 ; b$\leq$7

Voici une valuation convenant pour cette formule
a : 6.00
b : 0.00

On se rend bien compte que la valuation proposée ne satisfait pas toutes les contraintes de problem.lp, et satisfait encore moins la formule de base. On ne se sait donc pas trop comment corriger cette erreur vu qu'elle est propre à GLPK.solve. En cherchant un peu on se rend compte que le parser de Lp connais les symboles $<$ et $>$, mais que pour GLPK, ils se traduisent par des inégalités larges. Ainsi on a que des inégalités larges. On ne peut donc pas prendre de négation et donc encore moins avoir un résultat correct dans le cas général. Le créateur de GLPK a d'ailleurs répondu que cela n'était pas un bug mais une volonté dans la masure ou cela n'avait pas de sens pour un solveur d'optimisations de prendre des inégalités strictes.

Pour résouder ce problème, on implémente donc les inégalité stricte grâce aux larges : $a < b$ devient $a \leq b + \epsilon$ avec un ϵ bien choisi qui est de l'ordre de 10^{-6} car en le prenant plus petit cela cause des problèmes dans GLPK.
Dans cette même idée un autre problème est la gestion des contraintes arithmétiques :

Formule : $a * b < 2$ or $a + b < 2$

Contenu de problem.lp :

maximize 0 ; subject to $a * b < 2$; $a + b < 2$;

La formule n'est pas satisfiable

On s'interessera aux expressions arithmétiques dans l'extension, mais ce qu'il faut retenir c'est que GLPK n'aime pas les multiplications. Par exemple pour lui la formule $a * b < 4$ n'est pas satisfiable.

Et si tout ces problèmes ne paraissent pas assez fou... GLPK n'aime pas les nombres négatifs à droite. En effet la formule $a < 0 and b < 0$ renvoie soit (0,0) dans GLPK car il n'y a pas d'inégalités strictes, soit insatisfiable. En revanche $a+b-2 < 0$ renvoie (0,0) sans aucun problème. Et ce problème ne vient pas de notre parseur, en effet dans la version classique de GLPK, $a \leq -1$ n'est pas satisfiable.

Enfin, sur un plan plus anecdotique mais assez agaçant, GLPK retourne les résultats des valuations au centième près. Ainsi avec la formule $5a+b > 5$ on nous renvoie (0,5.00) car b vaut en fait 5.000001, mais GLPK renvoie juste 5.00 ce qui est assez inquiétant.

5 Extensions

5.1 Amélioration de l'algorithme naïf

L'algorithme actuel est relativement naïf dans la mesure ou dans le pire des cas il teste l'entiereté des 2^n valuations. De plus dans cette version de l'implémentation on utilise une fonction negate pour negationner les contraintes à chaque fois que l'on crée le problème. Ainsi, une première amélioration est la création de deux tables de hachages : ht_plus et ht_moins qui sont telles que pour tout x_i, $x_i \xrightarrow{ht\_plus} c_i$ et $x_i \xrightarrow{ht\_moins}$ negate(c_i). Cela évite donc de tout recalculer à chaque fois. On cherche maintenant à améliorer l'algorithme de façon considérable.

Un premier problème de notre algorithme est qu'il élimine une valuation par une valuation, alors que l'intuition nous dis que parmis les n contraintes qui sont incompatibles ensemble, il existe très probablement un sous ensemble $S \subset [1, ..., n]$ tel que $|S| < n$ et les contraintes de S sont incompatibles. Ainsi, en ajoutant une clause indiquant que on ne peut pas mettre les contraintes de S ensemble à true, on élimine d'un coup $2^{n-|S|}$ valuations ce qui entraine une amélioration considérable des la complexité.

On doit donc maintenant trouver une facon de déterminer un tel ensemble S, de préférence de taille minimale, et calculable efficacement. Ainsi, si les n contraintes sont incompatibles, on lance l'algorithme rendant un sous ensemble incompatible plus petit.

Une méthode naïve pour cela reviendrait à parcourir les n contraintes une a une : enlever la contrainte i, si on devient compatible on la remet, sinon on l'enlève définitivement. On prouvera en annexe que cette méthode est moins efficace que celle décrite ci-dessous.

5.1.1 Dans la littérature

Le problème que l'on essaye ici de résoudre est celui de "Minimum Unsatisfiable Core" (MUC), c'est un problème difficile, mais dans la mesure où on a pas besoin d'avoir un ensemble minimal mais juste petit, on étudie une variation de ce problème : "Small Unsatisfiable Core". Il y a des outils qui le font plus ou moins bien comme Z3 développé par microsoft research. De manière générale ces algorithmes se basent sur l'algorithme prouvant l'incompatibilité de l'ensemble pour en déduire l'ensemble minimum. Or comme on utilise ici GLPK et pas un solver fait à la main, on a un oracle et on a donc pas accès aux résulats intermédiaire de GLPK. On ne peut donc pas essayer d'appliquer ce genre de méthodes.

5.1.2 La méthode des S_i

Une première façon de traiter ce problème est de déterminer par dichotomie le plus petit m tq $S_1 = \{c_1, ..., c_m\}$ est incompatible. Pour améliorer la taille de S_1, on peut le raffiner, pour ce faire on

peut essayer d'enlever des contraintes. On sait que pour tout $m' < m, S' = \{c_1, ..., c_{m'}\}$ est compatible, on détermine donc par dichotomie le plus grand k tel que $S = \{c_k, ..., c_m\}$ est incompatible.

Une idée serait de visualiser la liste comme une liste circulaire de taille n et de changer le premier point pour avoir différents ensembles S_i où S_i est l'ensemble obtenu en appliquant l'algorithme ci-dessus sur la liste commençant par i. On va créer un algorithme qui obtient tous les $S_i = \{c_i, ..., c_{m_i}\}$ possibles.

Proposition 1
Soit $i \in [1, n]$ *et* $S_i = \{c_k, ..., c_m\}$
$\forall\ i \leq i' \leq k, S_{i'} = S_i$

Preuve
On prend $S_{i'} = \{c_{k'}, ..., c_{m'}\}$. Comme S_i est incompatble, $\{c_{i'}, ..., c_m\}$ est incompatible, on a donc $m' \leq m$ par déinition du minimum. On a ainsi que $\{c_i, ..., c_{m'}\}$ est incompatible car $S_{i'}$ l'est et $S_{i'} \subset \{c_i, ..., c_{m'}\}$. Ce qui implique $m' \geq m$ par définition de m et donc $m = m'$.
De plus, on a $k' \geq k$, ainsi $S_{i'} \subset S_i$. Or par maximalité de k dans la la construction de S_i, on a $k' = k$, et donc $S_{i'} = S_i$.

Remarque
Soit $i \in [1, n]$ et $S_i = \{c_k, ..., c_m\}$ et $S_{k+1} = \{c'_k, ..., c'_m\}$, par définition de k, $\{c_{k+1}, ..., c_m\}$ est compatible. Donc $m' > m$

On considère la liste comme liste circulaire, et on a une variable *start* correspondant au début de la liste et r correspondant à là où on commence les recherches de m_{start}. Ainsi, à chaque étape, $start \leftarrow k + 1$ et $r \leftarrow m$. On effectue cela jusqu'à ce que $k > n$. Cela correspond à l'algorithme 2.

Algorithme 2 : Méthode des S_i par dichotomie

Entrées : L Une liste de n contraintes incompatibles
Sorties : S un ensemble incompatible
$start \leftarrow 1$
$r \leftarrow 1$
$card_{min} \leftarrow n$
$S_{min} \leftarrow L$
tant que $start \leq n$ **faire**
  $k, m \leftarrow$ incompatible_dicho(L,start,r)
  $start \leftarrow k + 1$
  $r \leftarrow m$
  si $m - k < card_{min}$ **alors**
    $card_{min} \leftarrow m - k$
    $S_{min} \leftarrow \{c_k, ..., c_m\}$
  fin
fin
return S_{min}

On obtient donc ainsi le plus petit des S_i. En notant $S_i = \{c_{k_i}, ..., c_{m_i}\}$. Cet algorithme effectue $\mathcal{O}(\frac{n}{\mathbb{E}(k_i - i)} log(n - (\mathbb{E}(m_i - i)))$ appels à l'oracle GLPK_solve. En effet, on applique l'algorithme de dichotomie en prenant en compte que le nouveau m sera plus grand que l'ancien, d'où le facteur $log(n - \mathbb{E}(m_i - i))$. Et comme on applique le nouveau *start* vaut $start + k$ on applique $\frac{n}{\mathbb{E}(k_i - i)}$ fois l'algorithme.

Enfin, l'application de cet algorithme nous permet d'éliminer $2^{n - \mathbb{E}(|S_i|)}$ valuations. Ce qui est beaucoup mieux que de les éliminer une par une.

5.1.3 Une amélioration des S_i

Dans l'algorithme actuel, on renvoie le plus petit S_i, mais on pourrait raffiner encore plus les S_i. En effet, soit $S_i = \{c_k, ..., c_m\}$, à l'issue de l'algorithme on obtient $\forall\ k' > k, m' < m, \{c_{k'}, ..., c_{m'}\}$ est compatible. Cependant cela n'exclus pas qu'il existe $S' \subset S, S' \neq S$ incompatible. On a en revanche démontré que tout S' de la sorte contiendra k et m sinon $\{c_{k+1}, ..., c_m\}$ ou $\{c_k, ..., c_{m-1}\}$ serait incompatible.

L'idée est donc de poser $S = \{k, m\} \cup T$, on veut déterminer le T minimal. Pour faire cela, il suffit de considérer la liste $L = [c_k, c_m, c_{k+1}, ..., c_{m-1}]$ et d'appliquer une variation de incompatible_dicho dessus. On note $m' = \min\{t, \{c_k, c_m, c_{k+1}, ..., c_t\}$ incompatible $\}$. Si $m' = m$ alors on renvoie $S' = \{k, m\}$.

Sinon, on pose $k' = \max\{t, \{c_k, c_m, c_t, ..., c_{m'}\}$ incompatible $\}$, et on renvoie $S = \{c_k, c_m, c_{k'}, ..., c_{m'}\}$. On se rend ainsi compte ici que de la même manière que pour la première itération, on a que tout $\forall S' \subset S$ tel que S' incompatible, alors $\{k, k', m', m\} \subset S'$. On peut alors réappliquer l'algorithme pour raffiner encore plus. On s'arretera lorsque $\{k, k', k'', ..., m'', m', m\}$ sera incompatible. Cet algorithme est écrit et prouvé en annexe.

Cette algorithme revient donc à prendre la dichotomie itérée plutôt que la dichotomie de base pour le calcul des S_i, et est écrit en annexe 6.2 . Ainsi, le calcul d'un S_i se fait en $\mathcal{O}(p * log(n))$ où p correspond au nombre d'itérations avant de trouver l'ensemble S_i. Le calcul du S_i minimal se fait donc en $\mathcal{O}(\frac{n}{\mathbb{E}(k_i - i)} * \mathbb{E}(p) * log(n - \mathbb{E}(m_i - i))$.

De plus, on prouve en annexe 6.3 que en notant R le resultat de l'algorithme de calcul des S_i par dichotomie itéree, on a $|R| \leq n * (1 - \frac{1}{|S_{min}|}) + 2$. Et cette borne peut sûrement être bien plus fine car elle est valable pour l'algorithme de calcul des S_i par méthode naïve.

5.1.4 Limite de cette méthode

Les méthodes qui se reposent qui la création des S_i ne sont pas optimales, notamment car il n'y a pas de raison que le meilleurs ensemble incompatible contenant c_i soit inclus dans $[c_i, c_{m_i}]$, la proposition suivante illustre ce fait.

Proposition 2
Soit $S_{min} \subset [1, n]$ le plus petit ensemble incompatible,
On atteint pas forcément S_{min} via les algorithmes basés sur les S_i.

Preuve
On va créer un contre exemple,

$\{c_1, ..., c_n\}, \{c_{1'}, ..., c_{n'}\}, \{c_{1''}, ..., c_{n''}\}$ compatibles tels que

$\forall i, c_i, ..., c_n, c_{1'}, ..., c_{(i)'}$ tel que $c_{(i)'}$ est le premier tel que c'est incompatible et cet ensemble ne peut pas être raffiné. Même relation entre les $c_{i'}$ et $c_{i''}$ et les c_i et $c_{i''}$.

On pose maintenant $\bar{c}$ tel que $\{c_1, \bar{c}\}$ incompatible et $\{\bar{c}, c_{1''}, ..., c_{n''}\}$ incompatible
Enfin, on défini $L = [c_1, ..., c_n, c_{1'}, ..., c_{n'}, \bar{c}, c_{1''}, ..., c_{n''}]$

Ainsi, si on applique les algos basés sur les S_i,
Si la liste commence par c_i, alors $S_i = \{c_i, ..., c_n, c_{1'}, ..., c_{(i)'}\}$.
Si la liste commence par $c_{i'}$ alors $S_i = \{c_{i'}, ..., c'_n, c_{1''}, ..., c_{(i)''}\}$.
Si la liste commence par $c_{i''}$ alors $S_i = \{c_{i''}, ..., c''_n, c_1, ..., c_{(i)}\}$.
Enfin, Si la liste commence par $\bar{c}$ alors $S_i = \{\bar{c}, c_{1''}, ..., c_{n''}\}$.

Ainsi, $\forall i, |S_i| \geq n$ et pourtant, $|S_{min}| = 2$

5.1.5 Solution Exacte

Une façon de déterminer la solution exacte est de calculer tous les sous ensembles possibles, par cardinal croissant et de vérifier s'il sont incompatibles ou non. Ainsi appeler l'oracle $\sum_{n=1}^{|S_{min}|} \binom{n}{k} = 2^{|S_{min}|+1}$ fois ce qui ne fait rien gagner par rapport à l'algorithme de base car on enlève moins d'une valuation par appel. Une autre façon serait de chercher $|S_{min}|$ par dichotomie, ce qui demanderait moins de calcul, avec de l'ordre de $2^{\frac{|S_{min}|}{2}}$ appels à l'oracle. Cependant, cela revient toujours à moins qu'avec une heuristique efficace. En effet, même si on obtient S tel que $|S| = 2 * |S_{min}|$, alors on enlève $2^{\frac{|S_{min}|}{2}}$ valuations pour un calcul polynomial. Ainsi, il est bien plus efficace de trouver des heuristiques polynomial que d'essayer de trouver la solution exacte dans la mesure où ce problème semble NP-dur.

Enfin, même si la méthode des S_i permet d'avoir des performances agréables, ce n'est qu'une première intuition et on pourrait surement obtenir de bien meilleur performances avec des algorithmes génétiques ou basé sur des méthodes de branch-and-bound. Il faudrait d'ailleurs effectuer une analyse en terme de probabilité pour quantifier précisément les performances des différents algorithmes.

5.2 Formules générales

Dans la forme actuelle, le programme prend en entrée des formules sous forme normale conjonctive, cela est relativement contraignant, et on pourrait donc créer une fonction formule_to_cnf permettant de traduire toute formule en format cnf, que l'on pourrait donc ensuite envoyer dans notre programme. Cela permettrait donc d'étudier la satisfiabilité de toute formule.

Cette extension étant la moins intéressantes de toutes, nous avons choisit de ne pas la traiter par manque de temps.

5.3 Contraintes avec des Aexp

Comme dit précédemment, les formules actuelles sont des égalités ou inégalités formées par des variables et des constantes, sans opérations telles que $((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$. On cherche ici à rendre les formules plus générales en prenant des expressions arithmétiques au lieu des variables, permettant alors de gérer des formules de la forme $((a + 4b = 1) \vee (a = 2)) \wedge (c * a - b \geq 3) \wedge ((b \leq 2) \vee (c - b \geq 3))$

Pour raisonner sur ce genre d'expressions, on reprend le parser déjà évoqué plus haut et on essaye de contourner les problèmes liés au parser de LP, notamment le fait qu'il n'accepte pas les parenthèses.

Pour cela, dans l'arbre arithmétique, si on a un noeud dont un fils (f) n'est pas une feuille (valeur ou variable) on rajoute une contrainte munie d'une nouvelle variable x_{new} telle que $x_{new} \leftarrow f$ et dans l'arbre, (f) est remplacé par $Var\ x_{new}$. Il n'y a donc plus aucune parenthèses, mais il y a plus de variables et de contraintes a satisfaire.

Cette solution a cependant le mauvais goût de compliquer des cas qui étaient précédemment gérés. Puisque l'arbre de l'expression arithmétique ne se souvient pas des parenthèses, à chaque fils non feuille on rajoute des variables et des contraintes, donc une longue expression sans parenthèse sera traitée comme si elle en avait.

De plus, on a été confronté à un autre problème de GLPK : "a * b = c" où a et b sont des variables est transformé par le parseur interne de Lp en "[a * b] = c", avec les crochets... Or, GLPK ne gère pas les crochets ... ce qui rend impossible ou du moins compliqué le traitement des expressions avec des multiplications.

Au final, même si notre algorithme supporte n'importe quelle expression arithmétique, du fait des problèmes de GLPK, il est recommandé de multiplier avec des pincettes.

6 Conclusion

En conclusion, ce projet avait pour objectif de résoudre le problème de satisfiabilité modulo théorie dans le contexte de la théorie arithmétique. Pour cela, nous avons utilisé une approche basée sur la coopération entre le solveur SAT Glucose et le solveur d'arithmétique linéaire GLPK. Notre approche consistait à transformer la SMT-formule en un couple SAT-formule, hashtable, puis à utiliser Glucose pour générer des valuations satisfaisantes et GLPK pour vérifier leur cohérence arithmétique, étape qui fût relativement complexe dans la mesure où GLPK n'est pas parfait et entraîne des erreurs. Si une valuation n'était pas cohérente, nous avons ajouté une clause bien choisie à la SAT-formule pour trouver une nouvelle valuation.

Nous avons ainsi démontré que notre méthode est efficace pour résoudre des problèmes de satisfiabilité dans la théorie arithmétique. Cependant, elle reste relativement naïve et pourrait être améliorée. Des perspectives d'amélioration incluent l'utilisation de solveurs plus performants, l'ajout de techniques d'optimisation pour réduire le temps de calcul, ou encore l'extension de la méthode à d'autres théories.

En somme, ce projet nous a permis de comprendre les techniques de résolution de problèmes de satisfiabilité modulo théorie dans la théorie arithmétique et a ouvert la voie à de nombreuses perspectives de recherche intéressantes.

Enfin, l'approche utilisée ici est appelée off-line SMT dans la mesure où la coopération se fait tour pas tour, mais dans la littérature, le plus efficace semble est la version on-line, c'est à dire une coopération plus étroite et beaucoup plus complexe qu'il pourraît être interessant d'étudier.

7 Annexe

7.1 Efficacité de la méthode des S_i par rapport à la méthode naïve

Une première observation est de dire que la méthode naïve est loin d'être parfaite. Considérons c_0, c_1 incompatibles et $c_2, ..., c_n$ incompatibles, tels que c_0 et c_1 sont compatibles avec les autres. En suivant l'algorithme, comme enlever c_0 puis c_1 ne change rien à l'incompatibilité, on obtient à la fin $S = \{c_2, ..., c_n\}$.

Considérons l'ensemble de contraintes incompatibles $\{c_1, ..., c_n\}$, On note $k^1 = min\{i, \{c_i, ..., c_n\}$ incompatible$\}$, k^1 correspond à la première dichotomie prise en 1, avec m^1 fixé à n. En fait on se rend compte que l'ensemble construit par l'algorithme naïf correspond à l'ensemble des k^i : les k obtenus par dichotomie itérée, démarée en 1 et en fixant $m^i = n$. Ainsi, la méthode de a dichotomie itérée est plus rapide et a les mêmes défauts en terme de performance que l'algorithme naïf. La méthode des S_i par dichotomie itérée est donc bien plus performante encore.

7.2 Algorithme de dichotomie itérée

Proposition 3
L'algorithme 3 termine et est correct vis à vis de la définition donnée en 5.1

Preuve
Pour la terminaison, on se rend compte que à chaque itération, $|Inc|$ augmente strictement. Or pour $|Inc| = n$ on aura toutes les contraintes et on retournera forcément Inc car on aura que tout sous ensemble de Inc est compatible et on sait que E est incompatible, d'ou $m = n$ et donc on renvoie Inc.

Pour la correction, on a montré dans 5.1 que tout sous ensemble de E incompatible contient nécessairement Inc. Or si en appliquant dicho_incompatible on a $m = |Inc|$, cela signifie que Inc est incompatible, on ne peut donc pas raffiner plus et on renvoie donc Inc. Si on peut raffiner plus alors comme décrit en 5.1, on rajoute k^i et m^i dans Inc et on réapplique l'algorithme.

Ainsi, l'algorithme termine et est correct du point de vue de la définition donnée en 5.1 .

Algorithme 3 : Dichotomie Itérée

Entrées : Un deux ensemble Inc et Reste tels que $Inc \cup Reste$ est incomaptible
Sorties : S un ensemble incompatible

// Inc va contenir les éléments qui sont à mettre dans le résultats : les k^i, m^i où le i fait référence à l'itération et non pas la contrainte de départ.

$E \leftarrow Inc \cup Reste$
$k^i, m^i \leftarrow$ dicho_incompatible (E)
si $m^i = |Inc|$ **alors**
  return Inc
sinon
  $Inc \leftarrow Inc \cup \{k^i, m^i\}$
  $Reste \leftarrow Reste \backslash \{k^i, m^i\}$
  dicho_itérée (Inc,Reste)
fin

Algorithme 4 : Méthode des S_i par dichotomie itérée

Entrées : L Une liste de n contraintes incompatibles
Sorties : S un ensemble incompatible
$start \leftarrow 1$
$r \leftarrow 1$
$card_{min} \leftarrow n$
$S_{min} \leftarrow L$
tant que $start \leq n$ **faire**
  $k^1, m^1, S_i \leftarrow$ dichotomie_itérée(L,start,r)
  $start \leftarrow k^1 + 1$
  $r \leftarrow m^1$
  si $|S_i| < card_{min}$ **alors**
    $card_{min} \leftarrow |S_i|$
    $S_{min} \leftarrow S_i$
  fin
fin
return S_{min}

7.3 Borne de performances

On cherche ici à avoir une borne supérieur sur nos algorithme qui quantifie le pire des cas. On va tout d'abord analyser l'algorithme de calcul des S_i par méthode naïve. Pour en déduire une borne sur l'algorithme de calcul des S_i par dichotmie itérée :

En notant R le resultat de cet algorithme , on a $|R| \leq n * (1 - \frac{1}{|S_{min}|}) + 2$

Proposition 4
Soit S un ensemble de contraintes : $S = \{c_1, ..., c_n\}$,
Soit $\{U_k, k \in K\}$ l'ensemble des sous-ensembles incompatibles de S.
On pose $d_k = min\{i, c_i \in U_k\}$; $d = max\{d_k, k \in K\}$; $K_d = \{U_k, d_k = d\}$
Alors, l'appel de la méthode naive renverra un élément de K_d.

Preuve
Soit $i < d$, alors $\{c_{i+1}, ..., c_n\}$ est incompatible car les élément de K_d sont dedans. Ainsi, on va enlever c_i de la réponse finale.
Ainsi, en appelant R l'ensemble retourné par l'algorithme, R est incompatible et son premier élément est c_d. Donc $R \in K_d$.

Proposition 5
Soit S un ensemble de contraintes : $S = \{c_1, ..., c_n\}$,
Soit S_{min} l'ensemble minimum incompatible,
Alors l'algorithme de calcul des S_i par méthode naïve renvoie un ensemble R tel que
$|R| \leq n * (1 - \frac{1}{|S_{min}|}) + 2$

Preuve
On pose $S_{min} = \{c_{m_1}, ..., c_{m_r}\}$ et $\delta = max\{|c_{m_i} - c_{m_{i+1}}|\}$,
Soit i_0 tel que $\delta = |c_{m_{i_0}} - c_{m_{i_0+1}}|$
On va donc considérer $S_{m_{i_0}+1}$: c'est à dire à partir de la contrainte qui suit $c_{m_{i_0}}$ dans S.

On a alors $d_{S_{min}} = \delta - 1$, et donc $d \leq \delta - 1$
Ainsi, $|S_{m_{i_0}+1}| \leq n - (\delta - 1)$,
Or $\delta \geq \frac{n}{|S_{min}|} - 1$
Donc $|S_{m_{i_0}+1}| \leq n - \frac{n}{|S_{min}|} + 2$,

Et comme la reponse R de cette méthode est le plus petit des S_i,
On a $|R| \leq n * (1 - \frac{1}{|S_{min}|}) + 2$

Enfin, comme l'algorithme final est plus efficace que la méthode naïve, On a une borne similaire pour notre algorithme. En effet, trouver d revient à trouver m dans la liste miroir, et tout le reste ne fait que raffiné l'ensemble trouvé afin de réduire encore plus la borne.

On peut ainsi dire que l'algorithme de calcul des S_i par dichotomie itérée respecte cette borne (quitte à considérer la liste en miroir), et devrait même avoir une borne plus fine grâce aux étapes suivantes.

7.4 Formalisation des Problèmes

On appelle "minimal unsat core" un sous-ensemble S' de S tel que S' est insatisfiable et pour tout $s' \in S'$ $S' \setminus \{s'\}$ est satisfiable.

On appelle "minimum unsat core" le minimal unsat core de cardinal minimum.

Dans notre cas, dans la mesure où l'on se sert de GLPK comme oracle, et qu'on n'utilise pas le fait que l'on considère le problème de l'incompatibilité arithmétique, on se place dans une autre catégorie

de problème. On pourrait appeler cette variant de probmème "Blackbox" minimal unsat core.

Définitions :

Soit $S = \{c_1, ..., c_n\}$ un ensemble considéré en boite noire, c'est à dire que l'on en connaît pas les relations entre les différents éléments ni même les éléments.

Soit $P : \mathscr{P}(S) \to \{V, F\}$ une propriété stable, c'est à dire

Si $P(S) = F$ alors $\forall S'$, $P(S \cup S') = F$

Black Box Minimal Unsat Core : Trouver un sous-ensemble minimal S' de S tel que $P(S') = F$.

Black Box Minimum Unsat Core : Trouver le sous-ensemble minimal de cardinal minimum.

λ-Black Box Minimal Unsat Core Approximation : Trouver un sous-ensemble minimal S' tel que $|S'| \leq \lambda * |Smin|$.

Les deux premiers problèmes ne sont que des possibles généralisations de ceux connus, en considérant un ensemble en boite noire. Il y a même de grande chance que cela soit équivalent au problème classique.

Le troisième problème est celui qui nous interesse dans la mesure ou la recherche du minimum est est NP-dur, on cherche des bons algorithmes d'approximation. Le but est donc de trouver des algorithmes pour $1 \leq \lambda \leq \frac{n}{|S_{min}|}$.

Actuellement, la borne démontrée nous donne un algorithme avec $\lambda = \frac{n}{|S_{min}|} * (1 - \frac{1}{|S_{min}|}) + \frac{2}{|S_{min}|}$. Ce qui en plus de ne pas être extrêment performant, dépend de S_{min} ce qui n'est pas le but.