Combinaison de la propagation et de la décomposabilité pour la résolution de contraintes du premier ordre

Khalil Djelloul

Combinaison de la propagation et de la décomposabilité pour la résolution de contraintes du premier ordre

Khalil Djelloul

Laboratoire d'Informatique Fondamentale d'Orléans.
Bat. 3IA, rue Léonard de Vinci. 45067 Orléans, France.
khalil.djelloul@univ-orleans.fr

Résumé

Les contraintes du premier ordre (toute quantification, tout symbole logique) jouent un role fondamental en informatique et mathématiques. D'un coté théorique la conception d'un solveur de contraintes du premier ordre dans une théorie T est la preuve formelle de la complétude de T. D'un point de vue pratique, les contraintes du premier ordre offrent une grande expressivité qui permet de modéliser d'une manière naturelle plusieurs problèmes complexes de différents domaines de l'intelligence artificielle. Un des derniers résultats publiés autour de la logique du premier ordre est "la décomposabilité" : une propriété partagée par plusieurs théories du premier ordre qui permet de dégager une procédure de décision générale qui pour toute proposition (formule sans variables libres) produit soit *vrai* soit *faux* dans toute théorie décomposable. Malheureusement, cette procédure de décision ne permet pas de résoudre des contraintes du premier ordre contenant des variables libres. Ce genre de problèmes est connu sous le nom de *first-order constraint satisfaction problems*. Nous présentons donc dans ce papier, le premier solveur complet de contraintes du premier ordre dans toute théorie décomposable T. L'idée principale est de combiner une propagation de sous-formules logiques avec des propriétés subtiles de la décomposabilité. Notre solveur est composé de neuf règles de réécriture qui transforme toute contrainte du premier ordre φ (qui peut évidement contenir des variables libres) en une formule équivalente ϕ qui est : soit la formule *vrai*, soit la formule *faux*, soit une formule simple ayant au moins une variable libre et n'étant équivalente ni à *vrai* ni à *faux*. Nous montrons l'efficacité de notre solveur en résolvant des contraintes du premier ordre sur des arbres finis ou infinis contenant un grand nombre de quantificateurs imbriqués et comparons les performances obtenues avec ceux de notre tout dernier solveur de contraintes du premier ordre dédié aux arbres finis ou infinis. Ceci est le premier solveur de contraintes du premier ordre pour toute théorie décomposable T.

Abstract

First-order constraints play a fundamental role in computer sciences and mathematic. From a theoretical point of view, a first-order constraint solver in a theory T is the formal proof of the completeness of T. From a practical point of view, first-order constraints offer a big expressivity which enable to model many complex problems in the artificial intelligence world. One of the last result in the first-order logic field is the so called "decomposability" : a new property shared by many first-order theories and which enable to deduce a general decision procedure which for any proposition (formula without free variables) produces either true or false in any decomposable theory T. Unfortunately, this decision procedure is not enough when we want to express the solutions of a first-order constraint having free variables. These kind of problems are generally known as *first-order constraint satisfaction problems*. We present in this paper, not only a decision procedure but a full first-order constraint solver for decomposable theories. Our solver is given in the form of nine rewriting rules which transform any first-order constraint φ (which can possibly contain free variables) into an equivalent formula ϕ which is either the formula true, or the formula false or a simple solved formula having at least one free variable and being equivalent neither to true nor to false. We show the efficiency of our solver by solving complex first-order constraints over finite or infinite trees containing a huge number of

imbricated quantifiers and negations and compare the performances with those obtained using the most recent and efficient dedicated solver for finite or infinite trees. This is the first full first-order constraint solver for any decomposable theory.

1 Introduction

Les contraintes du premier ordre (toute quantification, tout symbole logique) jouent un role fondamental en informatique et mathématiques. D'un coté théorique la conception d'un solveur de contraintes du premier ordre dans une théorie T est la preuve formelle de la complétude de T. D'un point de vue pratique, les contraintes du premier ordre offrent une grande expressivité qui permet de modéliser d'une manière naturelle plusieurs problèmes complexes de différents domaines de l'intelligence artificielle. Un des derniers résultats publiés autour de la logique du premier ordre est la notion de *théories décomposables* que nous avons introduit dans [6]. En effet, nous avons défini une nouvelle propriété, partagée par plusieurs théories du premier ordre, qui permet de dégager une procédure de décision générale qui pour toute proposition (formule sans variables libres) telle

$$\exists u_2 \forall u_1 \exists u_3 \neg \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg(\exists w_1\, v_1 = g(w_1)) \wedge \\ \neg(\exists w_2\, u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right],$$

produit soit *vrai* soit *faux* dans toute théorie décomposable.

Malheureusement, la procédure de décision des théories décomposable est loin d'être suffisante si l'on veut résoudre des contraintes du premier ordre avec variables libres. En effet, nous détectons trois problèmes majeurs :

(1) Une procédure de décision peut uniquement décider si une formule sans variables libres est vraie ou fausse mais cette dernière se retrouve très vite incapable de simplifier des formules contenant des variables libres. En effet, si l'on modélise un problème donné P par la contrainte du premier ordre φ suivante dans une théorie T :

$$\neg \left[\exists v_1\, u = f(v_1) \wedge \left[\begin{array}{l} \neg(\exists w_1\, u = f(w_1) \wedge w_1 = v_1) \wedge \\ \neg(\exists w_2\, u = f(v_1) \wedge w_2 = f(v_1)) \wedge \\ \neg(\exists w_3\, u = f(v_1) \wedge v_1 = f(w_3)) \end{array} \right] \right],$$

alors la résolution du problème P veut dire : transformer φ en une formule simple équivalente - appelée *formule résolue* - à partir de laquelle on peut facilement extraire l'ensemble des valeurs de la variable libre u qui vont satisfaire la formule φ dans tous les modèles de la théorie T. Notre procédure de decision pour les théories décomposables [6] n'est pas capable de résoudre la formule φ et peut uniquement tester s'il existe au moins une solution au problème P en résolvant la formule $\exists u\, \varphi$, c'est-à-dire :

$$\exists u \neg \left[\exists v_1\, u = f(v_1) \wedge \left[\begin{array}{l} \neg(\exists w_1\, u = f(w_1) \wedge w_1 = v_1) \wedge \\ \neg(\exists w_2\, u = f(v_1) \wedge w_2 = f(v_1)) \wedge \\ \neg(\exists w_3\, u = f(v_1) \wedge v_1 = f(w_3)) \end{array} \right] \right],$$

et la réponse sera soit la formule *vrai* (donc le problème P a au moins une solution), soit la formule *faux* (donc le problème P n'a aucune solution).

(2) si l'on utilise notre procédure de décision sur une formule φ qui contient au moins une variable libre alors nous pouvons très bien obtenir une formule équivalente ϕ ayant au moins une variable libre mais étant équivalente soit vrai soit à faux quelque soit l'instantitations des variables libres. La formule finale appropriée de φ devrait donc être la formule *faux* ou la formule *vrai* au lieu de ϕ. Examinons ce phénomène très subtil à travers un exemple. soit Tr la théorie des arbres finis ou infinis (dont on a montré la décomposabilité dans [6]) et utilisons notre procédure de décision sur la formule φ suivante :

$$\neg(\exists y\, x = f(y) \wedge \neg(\exists z w\, x = f(z) \wedge w = f(w))). \quad (1)$$

Nous obtenons alors la formule finale ϕ suivante :

$$\neg(\exists y\, x = f(y) \wedge \neg(\exists z\, x = f(z))). \quad (2)$$

Le problème est que cette formule contient x comme variable libre mais pour toute instanciation de x la formule instanciée est vraie dans tout modèle de la théorie Tr. En effet, la formule ϕ équivalente à

$$\neg(\exists y\, x = f(y) \wedge \neg(\exists z\, x = f(y) \wedge x = f(z))),$$

c'est-à-dire à

$$\neg(\exists y\, x = f(y) \wedge \neg(x = f(y) \wedge (\exists z\, z = y))),$$

donc à

$$\neg(\exists y\, x = f(y) \wedge \neg(x = f(y))),$$

qui est clairement équivalente à *vrai*. Donc, la formule résolue de la formule initiale (1) devrait être la formule *vrai* au lieu de (2). Ceci est un bon exemple dans lequel nous montrons les limites des procédures de décision lors de la résolution de contraintes du premier ordre avec variables libres.

(3) Le troisième problème est que la complexité de notre procédure de décision est exponentielle en temps et espace pour la plupart des théories décomposables. Si l'on considère par exemple la théorie Tr des arbres finis ou infinis alors les performances de notre procédure de décision dans Tr sont très moyennes par rapport à notre tout nouveau solveur de contraintes du premier ordre sur les arbres finis ou infinis que nous avons présenté en 2008 dans [5]. L'explication est simple : la procédure de décision crée des combinaisons booléénnes à base de distributions très coûteuses en temps et espace alors que notre solveur [5] utilise un tas de propriétés très subtiles pour la simplification et résolution des contraintes d'arbres.

Des algorithmes beaucoup plus élaborés doivent être alors développés si l'on veut résoudre des contraintes du premier ordre dans n'importe qu'elle théorie décomposable. Bien entendu, notre but n'est pas de savoir s'il existe des solutions ou pas, mais d'exprimer l'ensemble de ces solutions sous la forme d'une formule du premier ordre φ qui est soit la formule *vrai* (donc le problème est satisfaisable pour toutes les valeurs des variables libres), soit la formule *faux* (donc le problème est insatisfaisable pour toutes les valeurs des variables libres), soit une formule résolue simple ayant au moins une variable libre et n'étant équivalente ni à vrai ni à faux. Les algorithmes qui sont capables de produire une telle formule φ sont connus sous le nom de *solveur de contraintes du premier ordre*.

Contributions : nous présentons dans ce papier non pas une procédure de décision mais le premier solveur de contraintes du premier ordre dans toute théorie décomposable T. Le papier est organisé en six sections suivies d'une conclusion. Cette introduction constitue la première section de ce travail. La section 2 est dédiée à un rappel de la logique du premier ordre et de la notion de théories décomposables. Nous présentons dans la section 3 les formules de travail : ce sont des formules structurées qui sont caractérisées par une notion de *profondeur*. Nous introduisons également les *box-checkers* : ce sont des boites noires qui vont nous permettre de réduire certaines formules de travail à *vrai* ou à *faux* même si ces dernières contiennent des variables libres. Dans la section 4, nous introduisons 9 règles de réécriture qui manipulent des formules de travail et qui transforment toute formule de travail de profondeur d en une conjonction de formules de travail de profondeur inférieur ou égale à 2 à partir de laquelle nous pouvons facilement extraire une formule résolue simple. L'idée principale de ces règles de réécriture consiste en : (1) un parcours "top-down" de propagation de contraintes. Dans chaque niveau de formule de travail les conjonctions de formules atomiques sont propagés aux sous formules de travails imbriquées. Cette étape permet entre autres d'éliminer les formules qui contredisent leur sous formules en utilisant nos box-checkers. (2) un parcours "bottom-up" d'élimination de quantificateurs en utilisant des propriétés subtiles de la décomposabilité ainsi qu'une distributions très particulière. Nous présentons dans la section 5 notre solveur de contraintes du premier ordre pour toute théorie décomposable T. Ce dernier est à base des neufs règles de réécriture définies dans la section 4. Il transforme toute formule φ en une formule résolue. Nous montrons enfin dans la section 6 l'efficacité de notre solveur et comparons ses performances avec celles obtenues en utilisant notre procédure de décision [6] qui peut uniquement répondre par vrai ou par faux. Notre solveur est capable de résoudre des formules contenant plus de 80 quantificateurs imbriqués et alternés alors que notre procédure de décision sature la mémoire à partir de 40 quantificateurs. Nous comparons également les performances de notre solveur avec ceux de notre tout dernier solveur de contraintes du premier ordre dédié aux arbres finis ou infinis [5] et montrons que même si notre solveur est général pour toute théorie décomposable, il offre des performances très compétitives par rapport à un solveur dédié à une théorie décomposable précise.

Les box-checkers, les 9 règles de réécriture, le solveur et les benchmarks sont de nouvelles contributions dans ce papier. Par manque d'espace, nous avons mis toutes les preuves dans un appendice. Une version complète de ce travail est disponible en ligne à l'adresse `http ://khalil.djelloul.free.fr/jfpc2008.pdf`

2 Préliminaires

2.1 Logique du premier ordre

Soit V un ensemble infini de variables. Soit S un ensemble de symboles, appelé signature et partitionné en deux ensembles disjoints : l'ensemble F des symboles de fonction et l'ensemble R des symboles de relation. A chaque symbole de fonction et relation est associé un entier strictement positif n appelé *arité*. Un symbole n-aire est un symbole d'arité n. Une contrainte du premier ordre est une expression de l'une des onze formes suivantes :

$$\begin{array}{c} s = t,\ r(t_1, \ldots, t_n),\ \mathit{vrai},\ \mathit{faux}, \\ \neg\varphi,\ (\varphi \wedge \psi),\ (\varphi \vee \psi),\ (\varphi \rightarrow \psi),\ (\varphi \leftrightarrow \psi), \\ (\forall x\, \varphi),\ (\exists x\, \varphi), \end{array} \quad (3)$$

avec $x \in V$, r un symbole de relation n-aire pris dans R, φ et ψ des formules plus petites, s, t et les t_i des termes, c'est-à-dire des expressions de l'une des deux formes suivantes :

$$x, f(t_1, ..., t_n),$$

avec x pris dans V, f un symbole de fonction n-aire pris dans F et les t_i des termes plus courts. Les formules de la première ligne de (3) sont dites *atomiques*, et *à plat* si elles sont de l'une des formes suivantes :

$$\mathit{vrai},\ \mathit{faux},\ x_0 = x_1, x_0 = f(x_1, ..., x_n),\ r(x_1, ..., x_n),$$

où les x_i sont des variables éventuellement non-distinctes prise dans V, $f \in F$ et $r \in R$. On note AT l'ensemble des conjonctions de formules atomiques à plat.

Une occurrence d'une variable x dans une formule est dite *liée* si elle apparaît dans une sous formule de

la forme $(\forall x\,\varphi)$ ou $(\exists x\,\varphi)$. Elle est *libre* dans tous les autres cas. Les *variables libres* sont ceux qui ont au moins une occurrence libre dans cette formule. Une *proposition* est une formule sans variables libres.

Un *modèle* est un couple $M = (D, F)$, où D est un ensemble non vide d'individus de M et F un ensemble de fonctions et de relations dans D. On appelle *instanciation* d'une formule φ par des individus de M, la formule obtenue à partir de φ en remplaçant chaque variable libre x de φ par le même individu i de D et en considérant chaque élément de D comme un symbole de fonction d'arité 0.

Une *théorie* T est un ensemble de propositions éventuellement infini. On dit que le modèle M est un *modèle de* T, si pout tout élément φ de T, $M \models \varphi$. Si φ est une formule, on écrit $T \models \varphi$ si pour tout modèle M de T, $M \models \varphi$. Une théorie T est *complète* si pour toute proposition φ, une et une seule des propriétés suivantes est satisfaite : $T \models \varphi$, $T \models \neg\varphi$.

Soit M un modèle et T une théorie. Soient $\bar{x} = x_1 \dots x_n$ et $\bar{y} = y_1 \dots y_n$ deux mots sur V de même taille. Soient φ et $\varphi(\bar{x})$ deux formules. On écrit

$\exists\bar{x}\,\varphi$ pour $\exists x_1 ... \exists x_n\,\varphi$,
$\forall\bar{x}\,\varphi$ pour $\forall x_1 ... \forall x_n\,\varphi$,
$\exists?\bar{x}\,\varphi(\bar{x})$ pour $\forall\bar{x}\forall\bar{y}\,\varphi(\bar{x}) \wedge \varphi(\bar{y}) \rightarrow \bigwedge_{i\in\{1,\dots,n\}} x_i = y_i$,
$\exists!\bar{x}\,\varphi$ pour $(\exists\bar{x}\,\varphi) \wedge (\exists?\bar{x}\,\varphi)$.

Le mot $\bar{x}$, qui peut éventuellement être le mot vide ε, est appelé *vecteur de variables*. Sémantiquement, les nouveaux quantificateurs $\exists?$ et $\exists!$ signifient "au plus un" et "un et un seul".

Terminons cette sous-section par une notation commode concernant la priorité des quantificateurs : $\exists$, $\exists!$, $\exists?$ et $\forall$.

Notation 2.1.1 *Soit Q un quantificateur pris dans $\{\forall, \exists, \exists!, \exists?\}$. Soit $\bar{x}$ un vecteur de variables pris dans V. On écrit :*

$$Q\bar{x}\,\varphi \wedge \phi \;\; pour \;\; Q\bar{x}\,(\varphi \wedge \phi).$$

Exemple 2.1.2 *Soit $I = \{1, ..., n\}$ un ensemble fini. Soient φ et ϕ_i avec $i \in I$ des formules. Soient $\bar{x}$ et $\bar{y}_i$ avec $i \in I$ des vecteurs de variables. On écrit :*

$\exists\bar{x}\,\varphi \wedge \neg\phi_1$ *pour* $\exists\bar{x}\,(\varphi \wedge \neg\phi_1)$,
$\forall\bar{x}\,\varphi \wedge \phi_1$ *pour* $\forall\bar{x}\,(\varphi \wedge \phi_1)$,
$\exists!\bar{x}\,\varphi \wedge \bigwedge_{i\in I}(\exists\bar{y}_i\phi_i)$ *pour* $\exists!\bar{x}\,(\varphi \wedge (\exists\bar{y}_1\phi_1) \wedge ... \wedge (\exists\bar{y}_n\phi_n) \wedge vrai)$,
$\exists?\bar{x}\,\varphi \wedge \bigwedge_{i\in I}\neg(\exists\bar{y}_i\phi_i)$ *pour* $\exists?\bar{x}\,(\varphi \wedge (\neg(\exists\bar{y}_1\phi_1)) \wedge ... \wedge (\neg(\exists\bar{y}_n\phi_n)) \wedge vrai)$.

2.2 Théorie décomposable

Nous rappelons dans cette sous section la définition des *théories décomposables* que nous avons introduit dans [6]. Informellement, cette définition établie le fait que dans toute théorie décomposable T, chaque formule de la forme $\exists\bar{x}\,\alpha$, avec $\alpha \in AT$, est équivalente dans T à une formule décomposée de la forme $\exists\bar{x}'\,\alpha' \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\bar{x}'''\,\alpha'''))$ dans laquelle les formules $\exists\bar{x}'\,\alpha'$, $\exists\bar{x}''\,\alpha''$, et $\exists\bar{x}'''\,\alpha'''$ ont des propriétés élégantes qui peuvent être exprimées en utilisant les quantificateurs suivants : $\exists?$, $\exists!$ and $\exists_\infty^{\Psi(u)}$.

Dans tout ce qui suit, nous utiliserons l'abréviation svls pour *"sans variables libres supplémentaires"*. Une formule φ est équivalente à une formule ψ svls dans T signifie que $T \models \varphi \leftrightarrow \psi$ et que ψ ne contient pas d'autres variables libres que ceux de φ.

Définition 2.2.1 *Une théorie T est dite* décomposable *s'il existe un ensemble $\Psi(u)$ de formules ayant au plus u comme variable libre et trois ensembles A', A'' et A''' de formules de la forme $\exists\bar{x}\,\alpha$ avec $\alpha \in AT$ tels que :*

1. *Toute formule de la forme $\exists\bar{x}\,\alpha \wedge \psi$, avec $\alpha \in AT$ et ψ une formule quelconque, est équivalente dans T à une formule svls décomposée de la forme*

$$\exists\bar{x}'\,\alpha' \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\bar{x}'''\,\alpha''' \wedge \psi)),$$

   avec $\exists\bar{x}'\,\alpha' \in A'$, $\exists\bar{x}''\,\alpha'' \in A''$ et $\exists\bar{x}'''\,\alpha''' \in A'''$.
2. *Si $\exists\bar{x}'\alpha' \in A'$ alors $T \models \exists?\bar{x}'\,\alpha'$ et pour chaque variable libre y dans $\exists\bar{x}'\alpha'$, au moins une des propriétés suivantes est satisfaite :*
   - *$T \models \exists?y\bar{x}'\,\alpha'$,*
   - *il existe $\psi(u) \in \Psi(u)$ tel que $T \models \forall y\,(\exists\bar{x}'\,\alpha') \rightarrow \psi(y)$.*
3. *Si $\exists\bar{x}''\alpha'' \in A''$ alors pour tout x_i'' de $\bar{x}''$ on a $T \models \exists_\infty^{\Psi(u)} x_i''\,\alpha''$.*
4. *Si $\exists\bar{x}'''\alpha''' \in A'''$ alors $T \models \exists!\bar{x}'''\,\alpha'''$.*
5. *Si la formule $\exists\bar{x}'\alpha'$ appartient à A' et ne contient pas de variables libres alors c'est soit la formule $\exists\varepsilon vrai$, soit la formule $\exists\varepsilon faux$.*

Nous avons montré dans [6] la décomposabilité de plusieurs théories du premier ordre telles : theories des arbres finis ou infinis [10, 5], théorie équationnelle de Clark [2], théorie des rationnels et réels additifs ordonnés [8] et plusieurs autres combinaisons de ces théories [7]. A partir de la définition des théories décomposable nous avons déduit une procédure de décision qui pour toute proposition produit soit vrai soit faux. Cette procédure de décision est la preuve formelle de la complétude de toute théorie décomposable. Cependant, cette procédure de décision et loin d'être capable de résoudre des contraintes du premier ordre avec variables libres. Pour cela, nous présentons dans la section suivante un ensemble d'outils qui vont nous permettre de construire un algorithme de résolution de contraintes du premier ordre pour toute théorie décomposable.

3 Formules de travail et box-checkers

Soit T une théorie décomposable quelconque. Les ensembles $\Psi(u)$, A, A', A'' et A''' sont alors connus et fixés pour tout le reste de ce travail.

3.1 Formules normalisées

Définition 3.1.1 *Une formule normalisée φ de profondeur $d \geq 1$ est une formule de la forme*

$$\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i \in I} \varphi_i), \tag{4}$$

où I est un ensemble fini éventuellement vide, $\alpha \in AT$ et les φ_i des formules normalisées de profondeur d_i avec $d = 1 + \max\{0, d_1, ..., d_n\}$. De plus, toutes les variables quantifiées de φ ont des noms distincts et différents de ceux des variables libres.

Exemple 3.1.2 *Soient f et g deux symboles de fonction d'arité 1 pris dans F. La formule*

$$\neg \left[\begin{array}{l} \exists \varepsilon\ vrai \wedge \\ \left[\begin{array}{l} \neg(\exists x\, y = f(x) \wedge x = g(y) \wedge \neg(\exists \varepsilon\, y = g(x))) \wedge \\ \neg(\exists \varepsilon\, x = f(z)) \end{array} \right] \end{array} \right]$$

est une formule normalisée de profondeur égale à trois. La formule $\neg(\exists \varepsilon\ vrai)$ est une formule normalisée de profondeur 1.

Propriété 3.1.3 *Toute formule φ est équivalente dans T à une formule svls normalisée de profondeur $d \geq 1$.*

Nous avons montré cette propriété dans [6] où nous avons présenté un algorithme très simple qui transforme toute contrainte du premier ordre en une formule normalisée svls de profondeur $d \geq 1$.

Exemple 3.1.4 *Soit Cl la théorie équationnelle de Clark [2] et soit φ la contrainte du premier ordre suivante :*

$$\forall x\, \exists y\, x = y \wedge z = w \wedge \neg(x = w \vee \exists v\, w = v).$$

La formule précédente est équivalente dans Cl à la formule normalisée de profondeur 3 suivante :

$$\neg \left[\exists x\ vrai \wedge \neg \left[\exists y\, x = y \wedge z = w \wedge \left[\begin{array}{l} \neg(\exists \varepsilon\, x = w) \wedge \\ \neg(\exists v\, w = v) \end{array} \right] \right] \right].$$

3.2 Box-checkers

Nous introduisons maintenant une propriété qui utilise deux nouveaux outils logiques notés $BC1$ et $BC2$ (BC pour "box-checker") et qui va nous permettre de détecter si une formule normalisée de profondeur deux est équivalente à vrai ou à faux même si celle ci contient des variables libres.

Propriété 3.2.1 *Soit φ une formule normalisée de profondeur 2 de la forme :*

$$\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \neg(\exists\, \bar{y}_i\, \beta_i)),$$

avec $\alpha \in AT$ et $\beta_i \in AT$. Notons $\bar{z}$ le vecteur des variables libres de φ. Il est possible de tester si φ est équivalente à vrai ou à faux même si elle contient des variables libres. Pour cela il faut :

- *Calculer la valeur de vérité de la proposition $\forall \bar{z}\, \varphi$. Si le résultat est vrai alors φ est vraie dans T. Cette étape est réalisée par un Box-checker noté $BC1(\varphi)$.*
- *Calculer la valeur de vérité de la proposition $\forall \bar{z}\, \neg\varphi$. Si le résultat est vrai alors φ est fausse dans T. Cette étape est réalisée par un Box-checker noté $BC2(\varphi)$.*
- *Si $BC1(\varphi) = faux$ et $BC2(\varphi) = faux$ alors φ est ni vraie ni fausse dans T.*

Nous montrons que la complexité d'un tel algorithme est $n.Cpx$ si $n \neq 0$ et Cpx si $n = 0$, où Cpx est la complexité de l'algorithme de décomposition qui décompose toute conjonction quantifiée de formules atomiques à plat suivant la définition 2.2.1.

Les deux checkers $BC1$ et $BC2$ sont considérés dans ce papier comme deux boites noires : ils sont composés de deux algorithmes très techniques qui utilisent des propriétés complexes des théories décomposables. Par manque de place, nous avons jugé pas utile de s'attarder sur les détails de ces checkers et avons préféré détailler l'integralité du solver avec des exemples pertinents qui éclaircissent le fonctionnement général de notre algorithme (voir les sections 4 et 5). Cependant, le lecteur pourra trouver une version complète de ce papier avec tous les détails des checkers à l'adresse suivante : `http ://khalil.djelloul.free.fr/jfpc2008.pdf`

Exemple 3.2.2 *Soit Tr la théorie des arbres finis ou infinis [10, 5]. Testons si la formule φ suivante peut être simplifiée dans Tr à vrai ou à faux même si cette dernière contient x comme variable libre :*

$$\neg(\exists y\, x = f(y) \wedge \neg(\exists z\, x = f(z)))$$

Calculons $BC1(\varphi)$. D'après la propriété 3.2.1, $BC1(\varphi)$ est vrai si la valeur de vérité de la formule suivante est vraie dans Tr

$$\forall x \neg(\exists y\, x = f(y) \wedge \neg(\exists z\, x = f(z))).$$

Cette dernière est équivalente à

$$\forall x \neg(\exists y\, x = f(y) \wedge \neg(\exists z\, x = f(y) \wedge x = f(z))),$$

$$\forall x \neg(\exists y\, x = f(y) \wedge \neg(x = f(y) \wedge (\exists z\, z = y))),$$
$$\forall x \neg(\exists y\, x = f(y) \wedge \neg(x = f(y))),$$
$$\forall x \neg faux,$$

qui est finalement équivalente à vrai. Donc $BC1(\varphi) =$ *vrai et donc d'après la propriété 3.2.1, la formule* φ *est équivalente à vrai même si elle contient une variable libre. Cet exemple est celui présenté dans l'introduction de ce papier (voir la formule (1) page 3) et pour lequel nous avons noté que la procédure de décision de [6] ne peut pas détecter que* φ *est toujours vraie ou toujours fausse quelque soit la valeur de sa variable libre, et ne peut donc pas simplifier d'avantage la formule* φ*. Notre solveur peut résoudre entièrement* φ *et produit la formule résolue vrai grace aux box-checkers. Nous verrons également dans la section 6 pourquoi ces box-checkers augmentent considérablement les performances de notre solver par rapport à la procédure de décision de [6].*

3.3 Formules de travail

Nous allons maintenant introduire les formules de travail : ce sont des formules normalisées ayant un entier sur chaque négation. Cette légère modification va nous permettre de :

(1) lier un sens sémantique à chaque sous formule de la forme $\neg^k(\exists \bar{x}\, \alpha \wedge ...)$ d'après la valeur de l'entier k,

(2) contrôler l'exécution des règles de réécriture de notre solveur sur les formules normalisées (voir section 4).

Définition 3.3.1 *Une formule de travail est une formule normalisée dans laquelle toutes les occurrences de* $\neg$ *ont été remplacées par des* $\neg^k$ *avec* $k \in \{0, ..., 4\}$ *et telle que chaque occurrence d'une sous formule de la forme*

$$p = \neg^k(\exists \bar{x}\, \alpha \wedge q), \quad with \;\; k > 0, \tag{5}$$

satisfasse les k ***premières conditions*** *de la liste de conditions ci dessous. Dans (5) :* $\alpha \in AT$*,* q *est une conjonction de formules de travail de la forme* $\bigwedge_{i=1}^{n} \neg^{k_i}(\exists \bar{y}_i\, \beta_i \wedge q_i)$*, avec* $n \geq 0$*,* $\beta_i \in, AT$*,* q_i *une conjonction de formules de travail, et dans la liste de conditions ci dessous* $\exists \bar{x}'\, \alpha'$ *est la conjonction quantifiée de formules atomiques à plat de la sur-formule de travail*[1] p' *de* p *si elle existe.*

1. *Si* p' *existe alors* $T \models \alpha \rightarrow \alpha'$*.*
2. $BC1(\neg(\exists \bar{x}\, \alpha)) = faux$*.*
3. *Si* p' *existe alors* $BC1(\neg(\exists \bar{x}'\, \alpha' \wedge \neg(\exists \bar{x}\, \alpha))) = faux$*.*
4. *La formule* $\exists \bar{x}\, \alpha$ *appartient à* A'*.*

Nous insistons sur le fait que $\neg^k$ *ne signifie pas que la formule normalisée satisfait uniquement la* k *ème condition mais toutes les conditions* i *avec* $1 \leq i \leq k$*.*

Exemple 3.3.2 *La formule* $\neg^3(\exists x\, y = f(x) \wedge \neg^2(\exists z\, z = f(y) \wedge y = f(x))$ *est une formule de travail de profondeur 2 dans la théorie* Tr *des arbres finis ou infinis. En effet :*

- *Pour* $\neg^3$ *nous avons* $BC1(\neg(\exists x\, y = f(x))) = faux$ *car la formule* $\forall y\, \neg(\exists x\, y = f(x))$ *est fausse dans* Tr*.*
- *Pour* $\neg^2$ *on a* $Tr \models (z = f(y) \wedge y = f(x)) \rightarrow y = f(x)$ *et* $BC1(\neg(\exists z\, z = f(y) \wedge y = f(x))) = faux$ *car la formule* $\forall xy\, \neg(\exists z\, z = f(y) \wedge y = f(x))$ *est fausse dans* Tr*.*

Définition 3.3.3 *Une formule de travail est dite* initiale *si elle commence par* $\neg^3$ *et si toutes les autres négations de niveau inférieure sont de la forme* $\neg^0$*. Une formule de travail est dite* finale *si elle est de profondeur inférieure ou égale à deux et si toutes ces négations sont de la forme* $\neg^4$*.*

4 Transformation d'une formule de travail initiale en une formule de travail finale

Nous présentons dans la figure 1 neuf règles de réécriture qui transforment toute formule de travail initiale de profondeur d en une conjonction équivalente de formules de travail finales. Appliquer une règle $p_1 \Longrightarrow p_2$ sur une formule de travail p veut dire remplacer dans p une sous formule p_1 par la formule p_2, en considérant que le connecteur $\wedge$ est associatif et commutatif. Dans la figure 1, α, β et λ représentent des conjunctions de formules atomiques à plat, $\bar{x}$, $\bar{y}$ et $\bar{z}$ représentent des vecteurs de variables, q représente une conjonction de formules de travail, r représente une conjonction de : formules atomiques à plat et de formules de travail. I est un ensemble fini éventuellement vide[2]. **Toutes ces lettres peuvent être indicées et avoir des primes**.

Dans la règle (2), $BC1(\neg(\exists \bar{x}\, \alpha)) = vrai$. Dans la règle (3), $BC1(\neg(\exists \bar{x}\, \alpha)) = faux$. Dans la règle (4), $BC1(\neg(\exists \bar{x}\, \alpha \wedge \neg(\exists \bar{y}\, \beta))) = vrai$. Dans la règle (5), si $q' = vrai$ alors $BC1(\neg(\exists \bar{x}\, \alpha \wedge \neg(\exists \bar{y}\, \beta))) = faux$. Dans la règle (6) :

- La formule $\exists \bar{x}\, \alpha$ est équivalente dans T à une formule décomposée de la forme $(\exists \bar{x}'\, \alpha' \wedge (\exists \bar{x}''\, \alpha'' \wedge (\exists \bar{x}'''\, \alpha''')))$,

[1] En d'autres termes, p' est de la forme $\neg^{k'}(\exists \bar{x}'\, \alpha' \wedge p^* \wedge p)$ où p^* est une conjonction de formules de travail et p est la formule (5).

[2] Rappelons que si $I = \emptyset$ alors $\bigwedge_{i \in I} q_i$ est la formule *vrai*.

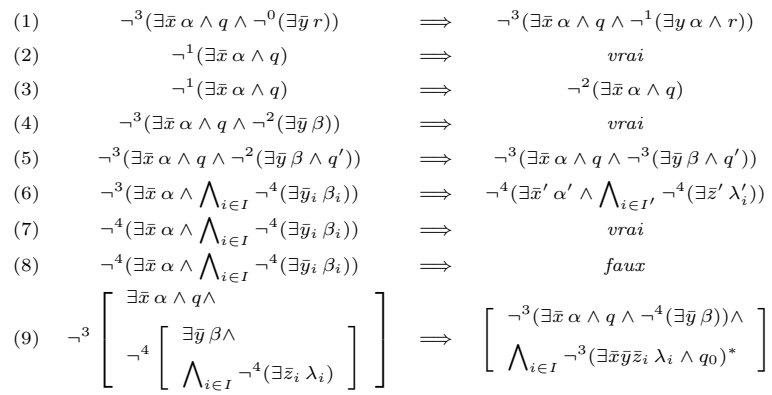

Fig 1. Transformation d'une formule de travail initiale en une formule de travail finale.

- la formule $\exists\, \bar{z}'\, \lambda'$ est la première partie de la formule décomposée de $\exists \bar{x}'''\bar{y}_i\, \alpha''' \wedge \beta_i$. En d'autres termes, la formule $\exists \bar{x}'''\bar{y}_i\, \alpha''' \wedge \beta_i$ est équivalente dans T à une formule décomposée de la forme $(\exists \bar{z}'\, \lambda' \wedge (\exists \bar{z}''\, \lambda'' \wedge (\exists \bar{z}'''\, \lambda''')))$. De plus, les variables quantifiées de chaque formule $\exists\, \bar{z}'_i\, \lambda'_i$ sont renommées par des noms distincts et différents de ceux des variables libres.
- I' est l'ensemble des $i \in I$ tels que $\exists \bar{y}'_i \beta'_i$ ne contienne pas d'occurrences libres d'une variable de $\bar{x}''$.

Dans la règle (7), $BC1(\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i\in I} \neg(\exists \bar{y}_i\, \beta_i))) = vrai$. Dans la règle (8), $BC2(\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i\in I} \neg(\exists \bar{y}_i\, \beta_i))) = vrai$. Dans la règle (9), $BC1(\neg(\exists \bar{y}\, \beta \wedge \bigwedge_{i\in I} \neg(\exists \bar{z}_i\, \lambda_i))) = faux$, $BC2(\neg(\exists \bar{y}\, \beta \wedge \bigwedge_{i\in I} \neg(\exists \bar{z}_i\, \lambda_i))) = faux$, l'ensemble I est un ensemble non vide et q_0 est la formule q dans laquelle toutes les occurrences de $\neg^k$ ont été remplacées par $\neg^0$. La formule $(\exists \bar{x}\bar{y}\bar{z}_i\, \lambda_i \wedge q_0)^*$ est la formule $(\exists \bar{x}\bar{y}\bar{z}_i\, \lambda_i \wedge q_0)$ dans laquelle nous avons renommé les variables de $\bar{x}$ et $\bar{y}'$ par des noms distincts et différents de ceux des variables libres.

Comment ça marche ? L'utilisation d'indices sur les négations des formules normalisées a pour but d'avoir un contrôle complet sur l'exécution des règles. En effet, nous avons construit nos règles de façon à ce qu'elles respectent tout le temps la stratégie suivante :

(i) un parcours "top-down" de propagation de formules atomiques à plat en suivant la hiérarchie arborescente des formules de travail et ce en utilisant les règles : (1),...,(5). Dans cette étape, les formules atomiques sont copiées dans toutes les sous-formules par la règle (1). Les formules inconsistantes ainsi que celles qui contredisent leurs sous-formules sont supprimées par les règles (2) et (4).

(ii) un parcours "bottom-up" d'élimination de quantificateurs et de réduction de profondeur par les règles : (6),...(9).

Plus précisément, à partir d'une formule de travail initiale φ de la forme $\neg^3(\exists \bar{x}\, \alpha \wedge \bigwedge_{i\in I} q_i)$ où tous les q_i sont des formules de travail dont les négations sont de la forme $\neg^0$, la règle (1) propage les formules atomiques de α dans les sous formules q_i, avec $i \in I$, et change la première négation de q_i en $\neg^1$. Les règles (2),...,(5) peuvent maintenant être appliquées. Les conjonctions de formules atomiques inconsistantes qui ont été créées après propagation sont supprimées par la règle (2). La règle (3) est ensuite appliquée et change la première négation de q_i en $\neg^2$. L'algorithme commence maintenant une nouvelle phase qui consiste à supprimer les formules qui contredisent leur sous formules en utilisant la règle (4). Notons que dans la règle (4), q est une conjonction de formules de travail de profondeur quelconque. Cette étape sera effectuée en utilisant le box-checker $BC1$ et permettra de réduire directement l'integralité de la formule de travail à vrai sans avoir à résoudre la sous formule de travail q qui peut avoir une très grande profondeur. La procédure de décision de [6] ne contient pas d'étape similaire et perd du temps et de l'espace considérable en résolvant la formule q par des distributions très coûteuses qui augmentent exponentiellement la taille des formules[3]. C'est pourquoi notre solveur est beaucoup plus efficace que cette procédure de décision. Une fois cette étape effectuée. La règle (5) est appliquée et change la deuxième négation en $\neg^3$. La règle (1) peut maintenant être appliquée une seconde fois car toutes les négations imbriquées à partir de ce niveau sont de la forme $\neg^0$. Cette boucle d'application de règles continue à tourner jusqu'à ce que les sous formules de travail de profondeur 1 deviennent de la forme $\neg^3(\exists \bar{y}_i\, \beta_i)$. La deuxième étape de notre solveur peut maintenant démarrer via l'application de la règle (6) avec $I = \emptyset$. Les négations des

[3] c'est la règle (5) dans [6].

formules de travail de profondeur 1 sont alors transformées en $\neg^4$. La règle (6) avec $I \neq \emptyset$ est encore une fois appliquée sur les formules de travail de profondeur 2 de la forme $\neg^3(\exists \bar{x}\, \alpha \wedge \bigwedge_{i\in I} \neg^4(\exists \bar{y}_i\, \beta_i))$ et produit des formules de la forme $\neg^4(\exists \bar{x}\, \alpha \wedge \bigwedge_{i\in I} \neg^4(\exists \bar{y}_i\, \beta_i))$. Les formules de travail inconsistantes de profondeur 2 ainsi que celles qui sont équivalentes à vrai sont réduites à vrai ou à faux par les règles (7) et (8). Notons que ces deux règles sont différentes des règles (2) et (4). En effet, nous pouvons construire plusieurs exemples dans lesquels les règles (2) et (4) ne peuvent pas être appliquées alors que la règle (7) est applicable. Une fois que toutes ces simplifications ont été faites, la règle (9) peut être appliquée sur les formules de travail de profondeur strictement supérieure à trois $\neg^3(\exists \bar{x}\, \alpha \wedge q \wedge \neg^4(\exists \bar{y}\, \beta \wedge \bigwedge_{i\in I} \neg^4(\exists \bar{z}_i\, \lambda_i)))$. Après chaque application de cette règle, de nouvelles formules de travail avec des négations de la forme $\neg^0$ sont créées ce qui implique l'exécution des règles de la première étape de notre solveur en commençant par la règle (1) et ainsi de suite. Après plusieurs applications de nos deux phases nous obtenons une conjonction de formules de travail dont la profondeur est inférieure ou égale à deux. Les règles sont alors appliquées une dernière fois jusqu'à ce que toutes les négations soient de la forme $\neg^4$. C'est une conjonction de formules de travail final.

Propriété 4.0.4 *Toute application répétée de nos règles de réécriture sur une formule de travail initiale termine et produit une conjonction équivalente svls de formules de travail finales.*

Exemple 4.0.5 *soient f et g deux symboles de fonction distincts pris dans F d'arité respectives $2, 1$. Soient w_1, w_2, v_1, u_1, u_2, u_3 des variables. Exécutons nos règles dans la théorie des arbres finis ou infinis sur la formule de travail initiale suivante :*

$$\neg^3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg^0(\exists w_1\, v_1 = g(w_1)) \wedge \\ \neg^0(\exists w_2\, u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right]. \quad (6)$$

D'après la règle (1), le formules précédente est équivalente dans T à

$$\neg^3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg^1(\exists w_1\, v_1 = g(w_1) \wedge v_1 = f(u_1, u_2) \wedge u_2 = g(u_1)) \wedge \\ \neg^0(\exists w_2\, u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right]$$

La règle (2) peut être appliquée sur la sous formule $\neg^1(\exists w_1\, v_1 = g(w_1) \wedge v_1 = f(u_1, u_2) \wedge u_2 = g(u_1))$. Donc, la formule précédente est équivalente dans T à

$$\neg^3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg^0(\exists w_2\, u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right],$$

qui, d'après la règle (1), est équivalente dans T à

$$\neg^3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg^1(\exists w_2\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right]$$

La règle (3) suivie de la règle (5) est appliquée. La formule précédente est donc équivalente dans T à

$$\neg^3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg^3(\exists w_2\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right].$$

La règle (6), avec $I = \emptyset$, est appliquée. Nous obtenons alors la formule équivalente suivante

$$\neg^3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg^4(\exists \varepsilon\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge u_1 = g(u_3)) \end{array} \right].$$

Encore une fois la règle (6) peut être appliquée, avec $I \neq \emptyset$. La formule précédente est donc équivalente à la formule de travail finale suivante

$$\neg^4 \left[\begin{array}{l} \exists \varepsilon\, u_2 = g(u_1) \wedge \\ \neg^4(\exists \varepsilon\, u_2 = g(u_1) \wedge u_1 = g(u_3)) \end{array} \right].$$

Nous avons vu dans l'exemple précédent comment les règles (1),...,(8) s'appliquent. Montrons maintenant sur un exemple simple comment utiliser la règle (9).

Exemple 4.0.6 *Soient s et 0 deux symboles de fonction pris dans F d'arité respective $1, 0$. Soient w_1, w_2, u, v des variables. Exécutons nos règles dans la théorie des arbres finis ou infinis sur la formule de travail suivante*

$$\neg^3 \left[\exists \varepsilon\, vrai \wedge \left[\begin{array}{l} \neg^4(\exists \varepsilon\, u = s(v)) \wedge \\ \neg^4(\exists w_1\, u = s(w_1) \wedge w_1 = s(v)) \wedge \\ \neg^4(\exists \varepsilon\, v = u \wedge \\ \left[\begin{array}{l} \neg^4(\exists \varepsilon\, v = u \wedge u = 0) \wedge \\ \neg^4(\exists w_2\, v = u \wedge u = s(w_2)) \end{array} \right]) \end{array} \right] \right]. \quad (7)$$

En considérons que

- $(\exists \bar{x}\, \alpha) = (\exists \varepsilon vrai)$
- $q = \left[\begin{array}{l} \neg^4(\exists \varepsilon\, u = s(v)) \wedge \\ \neg^4(\exists w_1\, u = s(w_1) \wedge w_1 = s(v)) \end{array} \right]$
- $(\exists \bar{y}\, \beta) = (\exists \varepsilon\, v = u)$
- $\bigwedge_{i\in I} \neg^4(\exists \bar{z}_i\, \lambda_i) = \left[\begin{array}{l} \neg^4(\exists \varepsilon\, v = u \wedge u = 0) \wedge \\ \neg^4(\exists w_2\, v = u \wedge u = s(w_2)) \end{array} \right]$

la règle (9) peut être appliquée et produit la formule de travail de la figure 2. A ce stade la, uniquement les règles (1),...,(8) peuvent être appliquées jusqu'à ce que toutes les négations soient de la forme $\neg^4$. La règle (9) ne pourra plus être appliquée car il n'y a pas de formule de travail de profondeur supérieure ou égale à 3.

5 Un solveur de contraintes du premier ordre pour toute théorie décomposable

Soit p une formule. Résoudre p dans T s'effectue de la manière suivante :

(1) Transformer la formule $\neg p$ (la négation de p) en une formule svls normalisée p_1 équivalente à $\neg p$ dans T. Pour cela, nous pouvons utiliser l'algorithme donné dans [6].

$$\left[\begin{array}{l}\neg^3(\exists\varepsilon\ vrai \wedge \neg^4(\exists\varepsilon\ u = s(v)) \wedge \neg^4(\exists w_1\ u = s(w_1) \wedge w_1 = s(v)) \wedge \neg^4(\exists\varepsilon\ v = u)) \wedge \\ \neg^3(\exists\varepsilon\ v = u \wedge u = 0 \wedge \neg^0(\exists\varepsilon\ u = s(v)) \wedge \neg^0(\exists w_{11}\ u = s(w_{11}) \wedge w_{11} = s(v))) \wedge \\ \neg^3(\exists w_2\ v = u \wedge u = s(w_2) \wedge \neg^0(\exists\varepsilon\ u = s(v)) \wedge \neg^0(\exists w_{12}\ u = s(w_{12}) \wedge w_{12} = s(v)))\end{array}\right]$$

Fig 2. Formule obtenue après application de la règle (9) sur la formule (7).

(2) Transformer p_1 en la formule initiale p_2 suivante

$$p_2 = \neg^3(\exists\varepsilon\ vrai \wedge \neg^0(\exists\varepsilon\ vrai \wedge p_1)),$$

dans laquelle toutes les occurrences de $\neg$ dans p_1 sont remplacées par $\neg^0$.

(3) Appliquer les 9 règles de réécriture sur p_2 jusqu'à ce qu'aucune règle ne puisse être appliquée. D'après la propriété 4.0.4 nous obtenons une conjonction p_3 svls de formules de travail finales de la forme

$$\bigwedge_{i=1}^{n} \neg^4(\exists\bar{x}_i\ \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg^4(\exists\bar{y}_{ij}\ \beta_{ij})).$$

Du fait que p_3 soit équivalent à $\neg p$ dans T, alors p est équivalente dans T à

$$\neg \bigwedge_{i=1}^{n} \neg(\exists\bar{x}_i\ \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\ \beta_{ij})),$$

qui est équivalente à la disjonction p_4 suivante

$$\bigvee_{i=1}^{n} (\exists\bar{x}_i\ \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\ \beta_{ij})).$$

Ceci est la réponse finale de notre solveur sur la contrainte p. Notons que les négations qui étaient aux début de chaque formule de p_3 ont été supprimées. De plus, la conjonction qui était au début de p_3 a été remplacée par une disjonction. Par conséquent, l'ensemble des solutions des variables libres de p_4 est tout simplement l'union des solutions de chaque formule résolue de la forme $\exists\bar{x}_i\ \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\ \beta_{ij})$. Grace aux deux checkers $BC1$, $BC2$ utilisées dans nos règles, nous montrons que si p_4 contient au moins une variable libre alors ni $T \models p_4$ ni $T \models \neg p_4$. En d'autres termes, p_4 ne peut être réduit ni à faux ni à vrai. Il ne reste plus qu'a présenter cette disjonction ayant uniquement un seul niveau de négation sous une forme explicite. Cette dernière phase dépend fortement de l'axiomatization et des propriétés propres à la théorie décomposable sur la quelle nous travaillons. Cette forme explicite sera par exemple une conjonction de formules atomiques à plat avec des membres gauches distincts si c'est la théorie des arbres et une conjonction d'équations atomiques mais non a plat si c'est la théorie des rationnels additifs. La procédure de décision donnée dans [6] produit des formules qui ne sont pas sous une forme aussi simple que p_4 (deux niveaux imbriqués de négations) et qui peuvent même être simplifiées d'avantage dans plusieurs cas en vrai ou en faux (voir la formule (2) à la page 3).

6 Benchmarks

6.1 Jeux à deux partenaires [4, 5]

Considérons le jeu à deux partenaires suivants : un couple (i, j) est choisi, avec i un entier non négatif (éventuellement nulle) et $j \in \{0, 1\}$. Chaque joueur joue une fois et change les valeurs de i et j suivant les règles suivantes :

- Si $j = 0$ alors le joueur actuel remplace i par $i - 1$ dans (i, j).
- Si $j = 1$ et i est paire alors le joueur actuel peut soit remplacer i par $i + 1$, soit remplacer j par $j - 1$, dans (i, j).
- Si $j = 1$ et i is impair alors le joueur actuel remplace i par $i + 1$ et j par $j - 1$ dans (i, j) ou remplace i par $i + 1$ dans (i, j)

Le première joueur qui ne peut plus jouer sans rendre i négatif a perdu. On peut représenter ce jeux par le graphe infini de la figure 3.

Il est clair que le joueur qui a atteint la position $(0, 0)$ et qui doit jouer a perdu. Supposons que c'est au tour du joueur A de jouer. Une position (n, m) est dite *k-gagnante* si, quelque soit la manière de jouer de l'autre joueur B, il est toujours possible au joueur A de gagner après avoir jouer au plus k coups. Nous avons montré dans [5] que pour tout entier k nous pouvons calculer toutes les positions k-gagnantes en résolvant une formule normalisée de profondeur $2k$ dans la théorie des arbres finis ou infinis. Pour cela, nous avons présenté dans [5] un algorithme efficace de résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis. Ce solveur utilise des propriétés très particulières valables uniquement dans la théorie des arbres et ne peuvent pas être généralisées à d'autres théories décomposables comme les rationnels additifs ordonnés.

Le temps d'exécution (CPU time en millisecondes) des formules $gagnant_k(x)$ sont donnés dans la table ci dessous ainsi qu'une comparaison avec ceux obtenus en utilisant notre tout dernier solveur de contraintes d'arbres [5]. Nous comparons aussi les performances

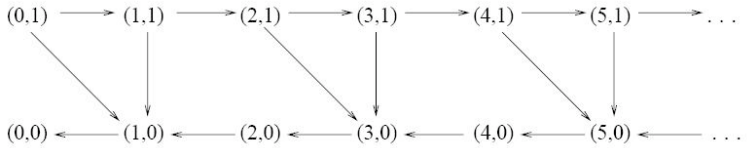

Fig 3. Représentation graphique de l'évolution de notre jeu à deux partenaires.

de nos 9 règles avec ceux obtenus en utilisant la procédure de decision des théories décomposables [6]. Les benchmarks sont effectués sur un 2 Ghz Pentium IV, avec 1024Mb de RAM. Le symbole "-" signifie mémoire saturée.

k ($gagnant_k(x)$)	1	4	5	10	20	40
C++ [6] (5 règles)	28	115	150	430	2115	–
C++ [5] (16 règles)	25	90	115	315	1490	15910
C++ (nos 9 règles)	27	98	133	353	1688	17124

La procédure de décision prend jusqu'à 25% de temps en plus par rapport à nos neuf règles et sature la mémoire pour $k > 20$, c'est-à-dire pour 40 quantificateurs imbriqués et alternés alors que notre solveur résout des formules ayant plus de 80 quantificateurs. D'autres part, nous atteignons les mêmes performances en espace et quasiment les mêmes performances en temps que notre tout dernier solveur d'arbres [5].

Nous expliquons maintenant pourquoi notre solveur est beaucoup plus efficace en temps et espace que notre procédure de décision des théories décomposables [6]. Cette dernière utilise plusieurs fois une distribution très particulière (la règle (5) dans [6]) qui diminue la profondeur des formules normalisées mais augmente exponentiellement le nombre de conjonctions des formules normalisées jusqu'à saturation de la mémoire. Notre solveur utilise une distribution similaire (règle (9)) mais uniquement après une phase de propagation de sous formules qui permet de tester s'il existe des formules qui contredisent leur sous formules ou qui contiennent des variables libres mais sont équivalentes à vrai ou à faux quelque soit l'instantiacion de ces variables libres (teste effectué par les box-checkers dans les règles (2), (4), (7) et (8)). Résoudre une formule $gagnant_k(x)$ dans notre jeux génère plusieurs formules normalisées qui contredisent leurs sous formules. Notre solveur supprime directement ces formules après la première étape de propagation alors que la procédure de décision ne peut pas détecter ce genre de formules et est obligée d'appliquer une distribution couteuse : après chaque application de cette règle, la profondeur diminue mais la taille de la conjonction augmente exponentiellement jusqu'à saturation de la mémoire. Voila pourquoi notre solveur est beaucoup plus performant en temps et espace que notre procédure de décision.

Il est important également de noter que les formules résolues obtenues par notre solveur ne peuvent jamais être simplifiée à vrai ou à faux. La procédure de décision des théories décomposables [6] n'est pas capable de garantir un tel résultat et nous a produit plusieurs fois au cours de nos benchmarks des formules avec variables libres mais pour lesquelles une simplification directe en vrai ou en faux est tout à fait possible.

6.2 Formules normalisées aléatoires

Nous avons également testé notre solveur sur des formules normalisées que nous avons générées aléatoirement dans la théorie des arbres finis ou infinis en respectant les conditions suivantes : pour toute sous formule normalisée de la forme $\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \varphi_i)$, où les φ_i sont des formules normalisées avec $n \geq 0$, nous avons :

- n est un entier positive choisi aléatoirement entre 0 et 4.
- Le nombre de formules atomiques à plat dans α est choisi aléatoirement entre 1 et 8. De plus, la formule atomique *vrai* apparaît au plus une fois dans α.
- Le vecteur des variables ainsi que les formules atomiques à plat de $\exists \bar{x}\, \alpha$ sont créés aléatoirement à partir d'un ensemble de 10 variables et 6 symboles de fonction : $f_0, f_1, f_2, g_0, g_1, g_2$. Chaque symbole de fonction f_j ou g_j est d'arité j avec $0 \geq j \geq 2$.

Les benchmarks sont réalisés sur un 2.5Ghz Pentium IV avec 1024Mb de RAM de la manière suivante : pour tout entier $1 \geq d \geq 41$ nous générons 10 formules normalisées aléatoires de profondeur d, on les résout et calculons enfin le temps moyen d'exécution (CPU time en millisecondes). Une fois encore, les performances (en temps et espace) de nos 9 règles sont impressionnantes par rapport à celles obtenues en utilisant la procédure de décision des théories décomposables. Ces résultats sont également très compétitifs par rapport à notre tout dernier solveur dédié aux arbres finis ou infinis [5]. Le symbole "-" ci dessous signifie saturation de la mémoire.

d	8	12	22	26	41
C++ [6] (5 rules)	375	1486	18973	–	–
C++ [5](16 rules)	202	504	3550	11662	2142824
C++ (our 9 rules)	221	612	4522	13654	2172632

7 Conclusion

Nous avons présenté dans ce papier le premier solveur complet de contraintes du premier ordre dans toute théorie décomposable T. La procédure de décision donnée dans [6] n'est pas capable de raisonner sur des formules du premier ordre avec variables libres et peut uniquement répondre par vrai ou par faux si l'on lui fournit une proposition en entrée. Notre solveur est sous forme de neuf règles de réécriture qui transforment toute contrainte du premier ordre φ en une contrainte résolue ϕ telle que ϕ est soit la formule *vrai*, soit la formule *faux*, soit une formule très simple, ayant au moins une variable libre et n'étant équivalente ni à vrai ni à faux

L'idée principale de notre solveur est de combiner une étape de propagation avec des propriétés subtiles de la décomposabilité à travers : (i) un parcours " top-down" de propagation de formules atomiques suivant la structure arborescente des formules normalisées. (ii) un parcours "bottom-up" d'elimination de quantificateurs et de réduction de profondeur des formules normalisées.

Nous avons également montré l'efficacité de notre solveur en comparant nos performances avec ceux obtenues en utilisant la procédure de décision des théories décomposables [6]. Nous arrivons ainsi à résoudre des formules avec plus de 80 quantificateurs alternés imbriqués alors que la procédure de décision sature la mémoire pour 40 quantificateurs. Notre solveur donne également des performances en temps et espace qui sont très compétitifs par rapport à notre tout dernier solveur dédié aux arbres finis ou infinis [5].

Actuellement, nous essayons de trouver une caractérisation des théories décomposables beaucoup plus simple que celle donnée dans [6]. Une des pistes intéressante est d'ajouter de nouveaux quantificateurs tels que $\exists^n$ *(il existe n)* et $\exists^{\Psi(u)}_{n,\infty}$ *(il existe n ou une infinité)*, afin d'augmenter la taille de l'ensemble des théories décomposables et de montrer une éventuelle équivalence entre théories décomposables et théories complètes ! Pour le moment nous ne disposons que de l'implication : théorie décomposable $\Longrightarrow$ théorie complète.

A ce jour, nous avons établi une longue liste de théories décomposables. On citera par exemple : la théorie de l'égalité, la théorie des arbres finis, la théorie des arbres infinis, la théorie des arbres finis ou infinis, la théorie des rationnels ou réels additifs ordonnés, la théorie de l'ordre dense sans extremes, la théorie des rationnels additifs ordonnés,...etc. Nous travaillons actuellement sur une nouvelles procedure de decision sur des listes [12] combinées à des arbres finis ou infinis.

Références

[1] Apt K. Principles of constraint programming. Cambridge University Press. 2003.

[2] Clark, K.L. 1978. Negation as failure. In Logic and Data bases. Ed Gallaire, H. and Minker, J. Plenum Pub.

[3] Colmerauer A., Dao T. Expressiveness of full first-order constraints in the algebra of finite or infinite trees, Journal of Constraints, Vol. 8(3) : 283-302. 2003.

[4] Dao, T. Resolution de contraintes du premier ordre dans la theorie des arbres finis ou infinis. Thèse d'informatique, Université de la mediterranée, France. 2000.

[5] Djelloul K., Dao T., Fruehwirth T. Theory of finite or infinite trees revisited. Theory and practice of logic programming (TPLP). 2008 (à paraitre).

[6] Djelloul K. Decomposable theories. Theory and practice of logic programming (TPLP). Vol 7(5) : 583-632. 2007.

[7] Djelloul K., Dao T. Extension into trees of first-order theories. In Proc of AISC'06 The 8th International Conference on Artificial Intelligence and Symbolic Computation. LNAI, vol 4120, pp. 53-67. 2006.

[8] Djelloul, K. About the combination of trees and rational numbers in a complete first-order theory. Proc of FroCoS'05 the 5th International Workshop on Frontiers of Combining Systems. LNAI, vol 3717, pp. 106-122. 2005.

[9] Fruehwirth T.,Abdennadher S. Essentials of Constraint Programming. Springer. 2003.

[10] Maher M. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In proc of LICS'88 Annual Symposium on Logic in Computer Science, pages : 348-357. 1988.

[11] Rybina, T., Voronkov, A. A decision procedure for term algebras with queues. ACM transaction on computational logic. 2(2) : 155-181. 2001.

[12] Spivey, J. A Categorial Approch to the Theory of Lists. In Proc of Mathematics of Program Construction, 375th Anniversary of the Groningen University, International Conference. LNCS, vol 375, pp. 399-408. 1989.

[13] Vorobyov, S. An improved lower bound for the elementary theories of trees. In Proc of CADE'96 the 13th International Conference on Automated Deduction. LNAI, vol 1104, pp. 275-287.1996.