DEA : Représentation de la Connaissance et Formalisation du Raisonnement.
Université Paul Sabatier Toulouse III, Année : 2003 - 2004

Coopération entre démonstrateurs par tableaux et SAT

par

Nicolas Troquard

Directeur de recherche : Olivier Gasquet

Résumé

Les méthodes SAT-based constituent une approche récente des procédures de décision pour les logiques modales. L'idée intuitive est que les procédures de décision en logiques modales propositionnelles peuvent être développées au-dessus d'une procédure de décision propositionnelle. Il est ainsi possible d'utiliser l'intégralité du foisonnant état de l'art sur le problème SAT (Davis-Putnam, transitions de phases...). KSAT est l'algorithme basique d'une telle procédure SAT-based pour le système K, qui peut être étendu à un large éventail de logiques. Nous projetons de nous inspirer de ces méthodes pour améliorer le démonstrateur générique **LoTREC** en le faisant collaborer avec un prouveur SAT. Pour cela, nous avons mis au point notre propre implémentation de KSAT et proposons quelques perspectives pour une future intégration.

Table des matières

Introduction

L'un des axes majeurs en intelligence artificielle concerne la représentation des connaissances et le raisonnement sur celles-ci. Le langage de la logique propositionnelle et celui de la logique modale propositionnelle sont des concepts à la fois simples et d'une grande puissance d'expression. De plus, des travaux anciens mais surtout récents ont permis de mettre au point des algorithmes de décision efficaces. Cela est d'autant plus vrai que ces dernières années ont vu l'apparition de systèmes hautement optimisés permettant de déterminer la satisfiabilité d'une formule en logique modale en s'appuyant sur un raisonnement propositionnel. Ces méthodes dites "SAT-based" sont issues de deux intuitions. La première est que le raisonnement en logique modale peut être implémenté comme une composition appropriée de raisonnements propositionnels. La seconde est que le raisonnement propositionnel peut être effectué très efficacement, en utilisant l'état de l'art foisonnant, des procédures de décision propositionnelles comme par exemple l'algorithme Davis-Putnam, et des avancées dans la connaissance du problème telles que les transitions de phases.

Plusieurs motivations convergentes sont à l'origine de notre démarche. De manière générale, les travaux de l'équipe LILaC à l'IRIT sur la formalisation de notions telles que connaissances, croyances, obligations et intentions, ainsi qu'interactions entre agents, ont amenés à développer des logiques modales qui nécessitent ensuite des expérimentations afin de les valider d'un point de vue réaliste. L'étude des procédures de démonstration automatique pour les logiques modales est une préoccupation traditionnelle de l'intelligence artificielle. Mais tandis que les démonstrateurs de la littérature ne concernent que des familles de logiques très restreintes, le démonstrateur automatique **LoTREC** actuellement développé dans l'équipe LILaC se veut un outil générique, avec un langage de programmation de stratégies qui est simple à manipuler.

Il est amplement souhaitable de tirer parti des progrès réalisés dans le cadre de la déduction automatique pour la logique propositionnelle. Ceci dit, l'intégration d'un démonstrateur SAT pourrait nuire à la flexibilité de **LoTREC**. Il s'agit de tirer profit des progrès dans le domaine SAT afin d'améliorer les démonstrateurs par tableaux existants, et d'étendre le démonstrateur **LoTREC** en le faisant coopérer avec un démonstrateur SAT pour gagner en performances sans renoncer à sa simplicité.

Nous sommes amenés à réaliser dans une première partie un état de l'art sur le problème de la satisfiabilité. En premier lieu pour la satisfiabilité en logique classique propositionnelle (SAT), décrivant en détail l'algorithme de décision le plus célèbre Davis-Putnam, et présentant les approches récentes d'étude de SAT, telles que les transitions de phases ou les heavy-tails, où résident les fondements de progrès dans la connaissance du problème. Nous abordons ensuite la satisfiabilité en logique modale (essentiellement dans le système $K(m)$). Nous nous intéressons en particulier aux procédures SAT-based, présentant l'algorithme basique KSAT en détail avec ses améliorations et ses extensions aux logiques normales et non-normales.

Dans une seconde partie nous abordons les aspects concrets de notre travail. Nous présentons pour cela le démonstrateur automatique **LoTREC**, puis notre implémentation de l'algorithme KSAT. Enfin, nous proposons des perspectives d'intégration et d'éventuels travaux futurs.

Partie I

La satisfiabilité : un état de l'art

Chapitre 1

La satisfiabilité en logique classique

Le problème de satisfiabilité d'une formule propositionnelle, consiste à déterminer s'il existe une affectation des variables de la formule, telle que celle-ci soit satisfaite. Nous présentons ce problème, ses techniques de résolution, ainsi que les principaux axes de la recherche actuelle dans le domaine.

Après un rappel rapide du problème SAT, nous présentons les algorithmes Davis-Putnam et GSAT. Nous présenterons ensuite les transitions de phase des instances SAT pour la satisfiabilité, la complexité des calculs et de la structure des solutions. Puis nous nous intéresserons aux distributions heavy-tails que l'on est emmené à rencontrer lors de l'étude de la complexité des recherches de solutions. Cela va nous permettre de mieux comprendre les mécanismes de la recherche de solutions à SAT, et ainsi pouvoir au mieux comprendre les stratégies d'amélioration.

1.1 Le problème SAT

Ce problème est le plus célèbre des problèmes NP-complets et de nombreux problèmes concrets peuvent être codés pour s'exprimer comme un problème de satisfiabilité. Coder un problème en SAT et le résoudre est parfois plus facile que développer un algorithme spécifique qui ne sera pas nécessairement plus efficace.

SAT constitue aussi un intérêt particulier pour l'IA puisqu'il est en étroite relation avec le raisonnement déductif : soit une base de faits Σ, on peut déduire α si et seulement si $\Sigma \cup \{\neg\alpha\}$ est inconsistant (ou insatisfiable).

Nous utiliserons le langage $\mathcal{L}_p$ suivant :

- $\mathcal{V}_p$, un ensemble fini ou dénombrable de variables propositionnelles (ou formules atomiques).
- $\mathcal{V}_{connect} = \{\neg, \wedge, \vee\}$, un ensemble de connecteurs logiques.
- $\mathcal{V}_{delim} = \{(,)\}$

Nous définissons $\mathcal{F}$ comme l'ensemble des formules (bien formées) sur $\mathcal{L}_p$.

Définition 1 *Une* interprétation *(ou* affectation*) est une fonction* $I : \mathcal{V}_p \rightarrow \{Faux, Vrai\}$.

Une interprétation définit la sémantique des formules atomiques. On peut l´étendre à $I : \mathcal{F} \rightarrow \{Vrai, Faux\}$ comme suit ($\alpha_i \in \mathcal{V}_p, 0 \leq i \leq n$) :

- $I(\alpha_0 \wedge ... \wedge \alpha_n) = Vrai \iff \forall i \in [0,n], I(\alpha_i) = Vrai$
- $I(\alpha_0 \vee ... \vee \alpha_n) = Vrai \iff \exists i \in [0,n], I(\alpha_i) = Vrai$
- $I(\neg\alpha_0) = Vrai \iff I(\alpha_0) = Faux$

Définition 2 *Soit une formule* ϕ*. Une interprétation est dite* complète *pour* ϕ *si elle est définie pour toute formule atomique de* ϕ*. Elle est dite* partielle *sinon.*

Définition 3 *Soit I une interprétation complète de ϕ et $I(\phi) = Vrai$, alors I est un* modèle *de ϕ, noté $I \models \phi$. ϕ est dite* valide *si toute interprétation complète de ϕ est un modèle de ϕ. ϕ est dite* satisfiable *ou* consistante *s'il existe au moins un modèle de ϕ.*

Définition de la satisifiabilité Le problème de satisfiabilité SAT se définit ainsi:

- instance: ϕ une formule de $\mathcal{F}$
- question: existe-t-il un modèle de ϕ?

Les formules en Forme Normale Conjonctive Nous nous ramènerons le plus souvent à des formules dans un format appelé *Forme Normale Conjonctive.*

Définition 4 *Un* littéral *est soit une formule atomique $\alpha \in \mathcal{V}_p$, soit sa négation $\neg\alpha$. α et $\neg\alpha$ sont dit* complémentaires.

Définition 5 *Une* clause *est une formule composée d'une disjonction de littéraux.*

Définition 6 *Une* base de clause *ou* formule en Forme Normale Disjonctive *noté* CNF *est une conjonction de clauses.*

Exemple 1 *Soient $\phi_1 = a_2 \wedge (a_0 \vee a_1 \vee \neg a_2) \wedge (\neg a_0 \vee a_2)$ et $\phi_2 = (a_0 \vee (a_1 \wedge \neg a_2)) \wedge (\neg a_1 \vee \neg a_2)$. ϕ_1 est une formule CNF, alors ϕ_2 ne l'est pas.*

Proposition 1 *Toute formule ϕ de $\mathcal{F}$ peut se traduire en une formule CNF ψ, telle que $\phi \iff \psi$, c'est-à-dire que pour une interprétation I, $I \models \phi \iff I \models \psi$.*

Nous pouvons alors définir le problème de la satisfiabilité ainsi :

- instance: $\mathbb{C}$ un ensemble fini de clauses sur $\mathcal{V}_p$
- question: existe-t-il une interprétation qui satisfasse l'ensemble des clauses $\mathbb{C}$?

1.2 Résoudre SAT

On peut regrouper les prouveurs en deux grandes catégories : les prouveurs complets et les prouveurs incomplets. Les derniers ne sont pas capables de prouver l'inconsistance d'une formule. Ils sont cependant utilisés pour trouver rapidement une solution pour les instances de grande taille. Nous présentons ici, une méthode de chaque catégorie. D'abord la procédure complète Davis-Putnam qui est une utilisation de la résolution à la recherche de consistance. Puis nous verrons GSAT, qui est un algorithme incomplet (effectuant une recherche gloutonne) fondé sur la recherche locale.

1.2.1 La procédure Davis-Putnam

La procédure Davis-Putnam (DP)[1] est un algorithme complet permettant de prouver la consistance d'une formule. Dans sa version la plus élémentaire, elle consiste à tester toutes les affectations de variables possibles et d'effectuer un retour-arrière (*backtrack*) lors de la rencontre d'un contre-modèle. Voyons d'abord quelques propriétés de simplification des formules logiques en Forme Normale Conjonctive (CNF). Nous les présentons en notation ensembliste.

Proposition 2 *Soient F une formule CNF et v une variable propositionnelle de F. Alors F est consistante si et seulement si $F \cup \{v\}$ est consistante ou que $F \cup \{\neg v\}$ est consistante.*

[1] aussi dénoté DPL, DLL ou DPLL pour Davis-Putnam-Logemann-Loveland.

L'intérêt est de supposer la valeur de vérité d'une variable v. On ajoute alors la clause unitaire $\{v\}$ à la formule si on suppose v vrai ou $\{\neg v\}$ si on la suppose fausse. Par la Proposition 3 on peut éventuellement simplifier la formule obtenue. Un algorithme basique DPLL pourrait se contenter d'utiliser cette règle et alors essayer toutes les affectations possibles.

Proposition 3 *Soient v une variable, et C une clause. Alors $\{\{v\}, \{v\} \cup C\}$ et $\{\{v\}\}$ sont équivalents; $\{\{v\}, \{\neg v\} \cup C\}$ et $\{\{v\}, C\}$ sont équivalents.*

Ce résultat induit la règle, dite de la *clause unitaire*, et signifie que si une formule F contient une clause unitaire $\{v\}$ alors toute clause de F contenant v ou $\neg v$ peut être simplifiée. On pourra aussi déterminer la valeur de v, qui sera $Vrai$ si la clause est positive et $Faux$ sinon. Cette règle de simplification, dont le coût est linéaire avec le nombre de littéraux, est très utilisée car elle améliore considérablement DPLL.

Proposition 4 *Soient F une formule CNF et v une variable de F telle que toutes les occurrences de v sont positives, (resp. toutes négatives). Alors F est consistante si et seulement si*

$$F' = F \setminus \{C : C \in F \text{ et } v \in C\} \text{ (resp. } F \setminus \{C : C \in F \text{ et } \neg v \in C\})$$

est consistante.

C'est la *règle du littéral pur*. Intuitivement, cela veut dire que lorsqu'une variable n'apparaît que positivement (resp. négativement) alors il suffit de lui affecter la valeur de vérité $Vrai$ (resp. $Faux$) pour satisfaire toutes les clauses ou elle apparaît. Ainsi, le problème se réduit à prouver la consistance de la base des clauses restantes. Cependant l'application de cette règle est très coûteuse et ne constitue pas nécessairement une bonne amélioration. C'est pourquoi nous ne la présentons qu'à titre informel et ne l'utiliserons pas dans l'application algorithmique.

Définition 7 *Une clause est dite :*

- positive *si aucun littéral négatif n'y apparaît*
- négative *si aucun littéral positif n'y apparaît*
- mixte *si au moins un littéral positif et au moins un littéral négatif y apparaissent*

En particulier, la clause vide est à la fois positive et négative, mais pas mixte.

Proposition 5 *Soit $F = \{C_1, ..., C_n\}$ une formule CNF contenant n clauses. Si pour tout i C_i est une clause négative, alors F est consistante. L'affectation de la valeur de vérité $Faux$ à toutes les variables, est un modèle de F. Par analogie, si pour tout i C_i est une clause positive, alors F est consistante, et l'affectation de la valeur de vérité $Vrai$ à toutes les variables est un modèle de F.*

Les propositions 2, 3 et 5 permettent de justifier l'algorithme de Davis-Putnam dans cette forme. La proposition 2 permet l'affectation *a priori* lors de l'appel récursif. La proposition 3 permet de justifier la routine *propagation_unitaire* et la proposition 5 autorise à conclure lorsque la formule ne contient aucune clause positive.

$Davis\_Putnam(\phi, \mu)$

Entrée :
ϕ : une formule propositionnelle en forme normale conjonctive
μ : une interprétation partielle
Sortie :
$Vrai$ si ϕ est satisfiable, $Faux$ sinon

.....

fonction $Davis\_Putnam(\phi, \mu)$
$(\phi, \mu) \leftarrow propagation\_unitaire(\phi, \mu)$;

si il existe une clause vide dans ϕ
alors retourne $Faux$;

si il n'y a plus de clause dans ϕ
alors retourne $Vrai$; /* μ *est solution* */

si ϕ ne contient aucune clause positive
alors retourne $Vrai$; /* μ *complété par l'affectation de tous les atomes de* ϕ *à* $Faux$ *est solution* */

$a \leftarrow choisi\_atome(\phi)$;
$Davis\_Putnam(\phi \wedge a, \mu)$;
$Davis\_Putnam(\phi \wedge \neg a, \mu)$;

La routine *propagation_unitaire* effectue les simplifications données dans la Proposition 3. On la retrouve dans la plupart des prouveurs basés sur l'algorithme DPLL.

$propagation\_unitaire(\phi, \mu)$

Entrée :
ϕ : une formule propositionnelle en forme normale conjonctive
μ : une interprétation partielle
Sortie :
(une simplification de ϕ en CNF, une mise à jour de μ)

fonction $propagation\_unitaire(\phi, \mu)$
tant que il existe une clause unitaire γ dans ϕ
si $\gamma = \{a\}$
alors $\mu' \leftarrow \mu \cup \{a \leftarrow Vrai\}$;
si $\gamma = \{\neg a\}$
alors $\mu' \leftarrow \mu \cup \{a \leftarrow Faux\}$;
$\phi' \leftarrow$ l'ensemble des clauses de ϕ ne contenant pas le littéral contenu dans γ;
$\phi'' \leftarrow \phi'$ dans laquelle on a supprimé tous les littéraux complémentaires à celui contenu dans γ;
$(\phi, \mu) \leftarrow (\phi'', \mu')$;
retourne (ϕ, μ);

1.2.2 La procédure GSAT

GSAT est une procédure incomplète de résolution des problèmes de satisfiabilité propositionnelle, proposée dans [SLM92]. Elle effectue une recherche locale gloutonne sur les affectations d'un ensemble de clauses propositionnelles. Au début de la procédure, on choisit au hasard une affectation des variables. Puis on change (*flips*) la valeur de vérité de la variable qui va permettre de maximiser l'augmentation du nombre de clauses satisfaites. On continue ainsi jusqu'à ce que l'on trouve la consistance ou que l'on atteigne MAX_FLIPS changements. On répète cela au plus MAX_TRIES fois.

$GSAT(\phi, MAX\_FLIPS, MAX\_TRIES)$

Entrée :
ϕ : une base de clauses
MAX_FLIPS : un entier
MAX_TRIES : un entier
Sortie :
$Vrai$ si un modèle de ϕ est trouvé, $Echec$ sinon

fonction $GSAT(\phi, MAX\_FLIPS, MAX\_TRIES)$
pour i de 1 à MAX_TRIES
$\mu \leftarrow$ une affectation de vérité générée aléatoirement;
pour j de 1 à MAX_FLIPS

si μ satisfait ϕ
alors retourne $Vrai$;
$a \leftarrow$ un atome tel qu'un changement de sa valeur de vérité dans μ maximise l'augmentation du nombre de clauses de ϕ satisfaites;
$\mu \leftarrow \mu$ avec la valeur de vérité de a inversée;
retourne $Echec$;

1.3 Vers un ensemble de modèles complet

Nous venons de voir deux algorithmes de recherche de modèles pour les formules propositionnelles. GSAT est une méthode incomplète, ce qui entraîne que (1) elle ne peut pas prouver l'inconsistance d'une formule, (2) si elle ne trouve pas de modèle, cela ne signifie pas que la formule n'est pas satisfiable. Au contraire, Davis-Putnam est une méthode complète, c'est-à-dire qu'elle trouvera un modèle à une formule si et seulement si cette formule est consistante. Néanmoins, elle ne permet pas de trouver *tous* les modèles d'une formule. En effet, la force de Davis-Putnam est de réduire l'espace de recherche. Lors de telles réductions, nous perdons des informations sur la formule originale.

Exemple 2 *Soit $\phi = a \wedge (a \vee b \vee \neg c)$. Par la règle de la clause unitaire nous allons affecter $Vrai$ à la variable a et propager, ce qui aura pour effet de résoudre la clause $a \vee b \vee \neg c$. Ainsi, seule l'interprétation partielle $\{a = Vrai\}$ sera trouvée, et sera complétée en un unique modèle $\{a = Vrai, b = Faux, c = Faux\}$ au moment de la sortie de la procédure.*

Or, certaines applications nécessitent d'avoir connaissance de l'ensemble *complet* des modèles.

Notation *On pourra convertir un modèle sous sa forme ensembliste en une formule bien formée, qui sera la conjonction des atomes évalués à $Vrai$ et de la négation des atomes évalués à $Faux$.*

Proposition 6 *Un ensemble de modèles $\{M_0, ..., M_n\}$ pour une formule ϕ est complet si et seulement si la formule $\psi = \phi \wedge \neg(\bigvee_{0 \leq i \leq n} M_i)$ est inconsistante.*

La proposition 6 met en évidence la méthode suivante, permettant de trouver l'ensemble complet des modèles finis d'une formule ϕ, sur le langage de la formule originale. Notons que cette méthode n'est utilisable qu'avec une procédure de décision complète.

- étape 0 : trouver un modèle M_0 de ϕ
- étape k : trouver un modèle M_k de $\phi \wedge \neg(\bigvee_{0 \leq i \leq k-1} M_i)$ (la formule ϕ en conjonction avec la négation des disjonctions des modèles trouvés aux étapes précédentes).

Si on ne trouve pas de modèle a l'étape N, cela signifie que l'on a trouvé tous les modèles finis de ϕ sur le langage de ϕ et il y en a N.

Exemple 3 *Soit $\phi = a \wedge (a \vee b \vee \neg c)$. On trouve le modèle $M_0 = a \vee \neg b \vee \neg c$. A l'étape suivante on cherche un modèle pour la formule $\phi \wedge \neg(a \wedge \neg b \wedge \neg c) = a \wedge (a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c)$. On trouve le modèle $M_1 = a \wedge \neg b \wedge c$. Et on continue de même. On trouve $M_2 = a \wedge b \wedge c$ et $M_3 = a \wedge b \wedge \neg c$. On ne trouve aucun modèle à la formule $\phi \wedge \neg(M_0 \vee M_1 \vee M_2 \vee M_3)$. Nous avons donc $\phi \iff (M_0 \vee M_1 \vee M_2 \vee M_3)$, et nous avons l'ensemble complet des modèles de ϕ.*

1.4 La complexité de SAT

Tout en gardant à l'esprit que SAT est un problème NP-complet, des questions naturelles se posent, quant à sa complexité en pratique. Quand est-il raisonnable d'utiliser une méthode complète? Est-ce que les cas difficiles apparaissent souvent? Quelle est sa complexité en moyenne? La grande difficulté à répondre à de telles questions est qu'il n'existe que peu de résultats analytiques sur la complexité de SAT. Pour l'instant, nous devons compter sur des résultats empiriques, qui eux sont foisonnants.

De nombreuses publications déclarent que le problème SAT n'est pas extrêmement complexe. Il a par exemple été suggéré que les instances SAT générées aléatoirement peuvent être aisément résolues en moyenne en $O(n^2)$. Cependant, pour examiner comment se comporte une procédure en moyenne, on doit considérer des instances générées suivant une distribution de probabilité. Cette complexité moyenne a donc été réfutée puisque dépendante du choix de la distribution.

1.5 Approches récentes

1.5.1 Les transitions de phases

L'étude des transitions de phases de SAT a émergée de l'intersection de la physique statistique, des mathématiques discrètes et de l'informatique théorique. En physique, les transitions se produisent dans les systèmes ayant de nombreux degrés de liberté, i.e. dans les systèmes spécifiés par des variables aléatoires nombreuses. Ces transitions représentent un changement d'état du système. Notre notion d'état pour une instance SAT sera sa *probable* satisfiabilité.

1.5.1.1 Les seuils de satisfiabilité

Introduisons d'abord la notion d'*ordre* et de *densité* pour les formules CNF.

Définition 8 *On appelle* ordre *d'une formule CNF ϕ le nombre de variables propositionnelles contenues dans ϕ.*

Définition 9 *Soit une formule CNF ϕ. On appelle* densité *de ϕ la valeur $\frac{\eta}{\omega}$ avec η le nombre de clauses de ϕ et ω son ordre.*

Il est aisé de concevoir qu'il sera difficile de satisfaire une instance ayant une forte densité, alors qu'une faible densité caractérisera une instance sous-contrainte et donc probablement satisfiable.

Il a été découvert au début des années 90 des valeurs de densité qui sembleraient marquer une frontière entre les instances satisfiables et celle qui ne le sont pas. Dans le cas des problèmes de 3-SAT aléatoires, des études expérimentales ont pu indiquer une densité seuil (*crossover point*) à $4,26$. Pour des instances de densité supérieures à $4,26$, la probabilité qu'elle soit satisfiable est proche de 0. Au contraire, en-deçà de ce seuil, une instance de 3-SAT aléatoire à toutes les chances d'être consistante[2]. Cependant, les tentatives pour prouver cette valeur mathématiquement sont restées vaines. Il semble que ce phénomène ne soit pas réservé aux seules instances de 3-SAT puisque des observations similaires ont été faites pour 4-SAT avec une valeur de seuil à $9,9$, pour 5-SAT avec un seuil autour de 21... Seule la valeur seuil de 1 a été prouvée pour 2-SAT [CR92].

En revanche de nombreux travaux ont été menés pour trouver rigoureusement un encadrement du seuil pour 3-SAT. Ainsi [DBM00] démontre une borne supérieure de $4,506$, et [AS00] une borne inférieure de $3,26$. Mais cela n'est pas suffisant pour prouver la transition rapide que semblent indiquer les expériences.

[2] Pour faire un parallèle avec la physique, une densité de $4,26$ est à l'état probable de la 3-satisfiabilité ce qu'est la température $0°C$ à l'état de l'eau.

Aussi concernant k-SAT en général, l'équation suivante a été formulée par Olivier Dubois :

$$\log(2) - x.2^{-k} - \exp(-x.k/(2^k - 1)) = 0$$

Pour $k = 3$ cette équation admet $4,252...$ pour solution ; pour $k = 4$ elle admet la solution $9,97...$ etc. Elle semble donc donner les valeurs constatées empiriquement. Mais la preuve n'en est pas faite.
Nous devons nous contenter d'une conjecture, presque unanimement acceptée :

Conjecture 1 *Pour tout $k \geq 2$, il existe une constante r_k telle que lorsque $n \to +\infty$, une formule* aléatoire *de k-SAT avec n variables et $r.n$ clauses est satisfiable avec une probabilité qui tend vers 1 pour tout $r < r_k$, et insatisfiable avec une probabilité qui tend vers 1 pour tout $r > r_k$.*

Notons tout de même l'existence d'un théorème de Ehud Friedgut [Fri99] qui montre l'existence d'une suite $r_k(n)$.

Théorème 1 *Pour tout $k \geq 2$, il existe une suite $r_k(n)$ telle que lorsque $n \to +\infty$, une formule* aléatoire *de k-SAT avec n variables et $r.n$ clauses est satisfiable avec une probabilité qui tend vers 1 pour tout $r < r_k(n)$, et insatisfiable avec une probabilité qui tend vers 1 pour tout $r > r_k(n)$.*

Prouver la conjecture 1 revient à prouver que $r_k(n)$ converge.

1.5.1.2 La complexité computationnelle

Nous appelons la complexité computationnelle d'une instance, le temps de calcul nécessaire pour décider de sa consistance. Certains auteurs évaluent cette complexité en utilisant le nombre d'appels récursifs d'une procédure de décision, cette mesure n'étant pas dépendante de l'environnement d'exécution. Plus généralement, on pourra parler de la complexité computationnelle d'une classe d'instance, qui sera le temps d'exécution moyen sur un échantillon de la classe.

La notion de transition de phase de satisfiabilité a été proposé comme outil pour l'étude de la complexité du problème SAT. Intuitivement, on peut imaginer qu'une formule de faible densité, et donc sous-contrainte, est aisément satisfiable et une preuve est vite trouvée. Comme il est facile de prouver l'inconsistance d'une formule ayant une forte densité, une contradiction étant vite trouvée. [SML96] avance que la densité d'une instance 3-SAT est intimement liée à sa complexité computationnelle. Autrement dit, une instance 3-SAT ne sera difficile à prouver qu'autour d'une densité de $4,26$. En fait, on obtient une courbe de complexité en fonction de la densité de la formule que l'on appelle *easy-hard-easy pattern*, le pic correspondant à la densité seuil.

Cependant, malgré les efforts permanents de la recherche dans ce domaine, les choses restent assez floues. En effet, [CDS+00] met en doute cette conclusion, en particulier à cause du protocole expérimental qui étudie la complexité en fonction de la densité de l'instance au lieu de sa taille (son ordre) comme il est d'usage en complexité computationnelle. Il y est présenté une expérimentation permettant de déterminer comment la complexité moyenne d'une instance 3-SAT (en fonction de l'ordre, pour une densité donnée) dépend de la densité, et si les observations dépendent du solveur utilisé.
Finalement [CDS+00] conclu qu'il n'existe pas de connexion entre la transition de phase en complexité computationnelle et le phénomène de seuil de satisfiabilité. Le passage de la complexité polynomiale à exponentielle ne se fait pas au point seuil, ni même autour comme déclaré dans [SML96]. De plus, la densité à laquelle la complexité passe de polynomiale à exponentielle varierait avec le choix du prouveur.

1.5.1.3 Structure des solutions de SAT

Nous avons vu que la densité d'une formule SAT était un paramètre d'ordre. Mais un concept plus profond pour les transitions de phase de SAT est le *backbone* qui a été suggéré comme étant un paramètre d'ordre plus pertinent pour caractériser un problème complexe.

Définition 10 *Le* backbone *d'une formule ϕ est l'ensemble des littéraux ayant la même valeur dans tous les modèles de ϕ.*

On trouve aussi dans la littérature des définitions indiquant que le backbone est l'ensemble des littéraux faux, ou l'ensemble des littéraux vrais dans tous les modèles.

Deux travaux indépendants ont été menés : [Zha01] qui est une étude empirique de la phase de transition de SAT à travers les backbones, et [XL99] qui est une étude analytique sur la similarité des solutions.

[Zha01] observe que :

- Quand la densité d'une formule est très grande, il y a peu de solutions.
- La taille du backbone d'une formule est d'autant plus grand que sa densité est grande.

En utilisant une notion de métrique fidèle à l'idée que deux interprétations sont d'autant plus proches qu'elles évaluent identiquement des variables, on peut en conclure que l'ensemble des solutions d'une instance grandement contrainte est clusterisé dans un petit voisinage.

[XL99] arrive à la même conclusion, introduisant le *degré de similarité majeur* : un paramètre décrivant la structure des solutions dont les valeurs montrent combien les différents modèles d'une formule sont proches les uns des autres. Il est prouvé formellement que lorsque la densité d'une formule atteint une certaine valeur, la structure des solutions subit une transition de phase, ce qui montre que les différentes assignations satisfaisant une formule passent soudainement de relativement différentes, à très similaires les unes des autres. Ceci est un autre type de transition de phase pour le problème SAT, à côté de celui de la satisfiabilité. Mais surtout, il présente un intérêt considérable pour l'amélioration des techniques de décision. En effet, l'appréhension de la structure des solutions au regard de la valeur de la densité d'une formule (triviale à calculer), permettrait à moindre coût de guider la recherche.

1.5.2 Les phénomènes heavy-tail

Voyons d'abord un exemple intriguant des probabilités. Un homme ivre qui marche, effectue à chaque pas un écart aléatoire à gauche ou à droite (parallèlement à un axe x), tous deux équiprobables et égaux. Alors, partant de l'origine, sa marche le ramènera sur l'axe de l'origine avec une probabilité de 1. En revanche, le temps qu'il lui faudra pour cela est infini, et, en moyenne, il va parcourir toutes les valeurs de l'axe x avant son retour.

Aussi, par intuition, si on imagine que cet homme passe l fois par l'origine en k pas, alors lors d'une marche m fois plus longue, on peut s'attendre à ce qu'il revienne à l'origine $m.l$ fois. Mais il peut être montré qu'il ne franchira l'axe que $\sqrt{m}.l$ fois en moyenne. Cela signifie qu'il existe des périodes étonnamment longues lors d'une marche aléatoire, entre deux retours à l'origine. En fait, lors d'une série de r marches aléatoires terminant chacune à l'origine, certaines marches vont en moyenne être du même ordre que la longueur de toutes les autres combinées, quelle que soit la valeur de r.

De tels phénomènes ne sont pas rares avec les distributions heavy-tail, et constituent d'ailleurs un aspect inhérent de celles-ci. Ces distributions sont donc de bons modèles pour les phénomènes présentant des fluctuations extrêmes [GSCK00].

1.5.2.1 Les distributions heavy-tail

Les distributions de probabilité standards, telles que les distributions normales, ont une décroissance exponentielle. Cela signifie que les évènements qui ne sont pas proche de la moyenne de la distribution sont très rares.

Les distributions heavy-tails sont des distribution ayant une décroissance polynomiale

$$\Pr\{X > x\} \sim Cx^{-\alpha}, x > 0$$

avec $0 < \alpha < 2$ et $C > 0$ des constantes.

Les distributions s'observent graphiquement comme une courbe linéaire lorsque l'on se place dans un repère $\log - \log$.

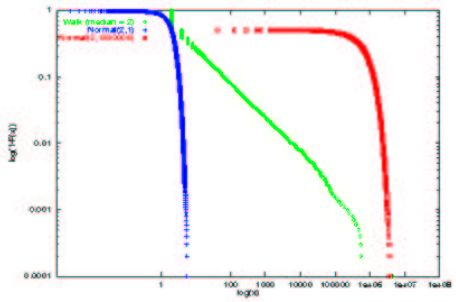

Figure 1.1: le comportement d'une distribution heavy-tail (exemple de la marche aléatoire), par rapport aux décroissances exponentielles de deux distributions normales. (Courbes tirées de [GSCK00].)

La figure 1.1 représente le complément de la distribution cumulée, soit $1 - F(X) = 1 - \Pr\{X \leq x\} = \Pr\{X > x\}$, avec $F(x)$ la distribution cumulée de $f(x)$. La fonction de probabilité $f(x)$ donnant la probabilité de retourner à l'origine en exactement x pas. Donc, $1 - F(x)$ nous donne la probabilité de retourner à l'origine en plus de x pas. La courbe $\log - \log$ de la marche aléatoire montre une décroissance polynomiale, donc $1 - F(X) = \Pr\{X > x\} \sim Cx^{-\alpha}$ et la distribution est une heavy-tail.

Cet exemple permet de voir comment mettre en évidence le comportement de problèmes ayant des fluctuations extrêmes, à l'image de SAT lors des transitions de phases, mais aussi aux recherches avec retour-arrière en général.

1.5.2.2 Les heavy-tails et SAT

Les comportements que l'on a vu peuvent modéliser les recherches avec retour-arrière, et signifient que des recherches très longues peuvent se produire bien plus souvent que l'on pourrait s'y attendre si nous avions affaire à des distributions de probabilité standards. De la même façon, il a été observé sur certaines instances de problèmes SAT suffisamment difficiles que des résolutions rapides se produisaient plus souvent que prévu. Cela montre un heavy-tail *à gauche* (c'est-à-dire pour les $X < x$, par opposition aux heavy-tails à droite pour les $X > x$) de la distribution. Ce phénomène a des conséquences significatives pour les stratégies à adopter, comme par exemples les *relances*.

Dans de nombreux problèmes à forte combinatoire, il est souvent beaucoup plus facile d'affecter les bonnes valeurs à certaines variables qu'à d'autres, les *variables critiques contraintes*, qui seraient la source des heavy-tails [GSCK00]. En revanche, une fois que celles-ci sont déterminées, la suite de la résolution est aisée. Par conséquent, la meilleure stratégie pour une procédure de recherche avec retours-arrières (telle que Davis-Putnam) est de se concentrer à trouver des valeurs qui satisfont l'ensemble des variables critiques. Les *backbones*, ensembles de littéraux ayant la même valeur pour toute solution, sont des variables critiques [SW01, Zha01, XL99]. Il se trouve que le nombre de ces variables critiques est lié aux transitions de phase.

Une approche récente sur la compréhension de ces heavy-tails est celle des *small-worlds*. [Wal99] observe qu'une topologie de small-world a un impact réel sur la résolution des problèmes de graphes, introduisant souvent des heavy-tails dans la complexité de résolution. [GSCK00] indique qu'une telle topologie est caractérisée par un petit nombre de contraintes globales, combiné avec un grand nombre de connexions locales. Dans de tels problèmes, les nombreuses interactions peuvent

définir les variables critiques du problème. Une application à l'étude de la complexité de SAT pourrait s'avérer intéressante.

1.6 Davis-Putnam ou GSAT?

Le problème se pose lorsque l'on souhaite faire une comparaison d'efficacité de deux méthodes. Les auteurs de GSAT avancent que leur procédure supplante DPLL en efficacité. C'est effectivement le cas sur des classes d'instances aléatoires difficiles. Les résultats exposés dans [SBH] semblent d'ailleurs confirmer que les algorithmes de recherche locale sont de loin plus efficaces que les méthodes *backtrack* sur la classe des instances aléatoires.

améliorations GSAT Notons que GSAT est très sensible aux minima locaux, mais aussi aux plateaux : lorsque la meilleure affectation ne change pas le nombre de clauses satisfaites. Quand cela arrive, les essais successifs avec une nouvelle instanciation initiale suffisent. Mais d'autres mécanismes permettent de quitter les minima locaux. Une amélioration consiste alors à introduire un choix de variable aléatoire lorsqu'il n'est pas possible d'augmenter le nombre de clauses satisfaites (consulter [SKC93] pour plus de détails). Les améliorations successives ont mené à un nouvel algorithme WSAT (pour Walk SAT) [SKC94].

améliorations DPLL Les règles dérivées des propositions 3, 4 et 5 données dans la section 1.2.1, sont déjà des améliorations de la procédure basique. Et même si la règle de la clause unitaire est (presque) toujours utilisée, elle n'est pas nécessaire.

Les solveurs qui utilisent l'algorithme de Davis-Putnam se distinguent souvent par leurs heuristiques. Particulièrement lors de l'écriture de *choisi_atome* qui permet de choisir une variable pour laquelle on va développer un arbre de recherche. Par exemple :

- la règle de la *plus petite clause* dit de prendre un littéral choisi aléatoirement dans une plus petite clause, éventuellement choisie aléatoirement.
- une autre heuristique consiste à choisir un littéral de la plus petite des clauses positives.

Aussi, l'observation faite en 1.5.2.2 sur les heavy-tails à gauche est particulièrement intéressante. En effet, cela signifie que les procédures de recherche sont en quelque sorte plus efficaces au début de la recherche. Cela suggère qu'une suite de courtes exécutions de la procédure s'avérerait plus efficace qu'une simple et longue exécution. [GSCK00] exhibe des résultats expérimentaux qui tendent à le prouver. L'utilisation de relances de la recherche éliminerait les distributions heavy-tail à droite des distributions. De façon très surprenante, de telles relances de procédure de recherche peuvent alors tirer parti des heavy-tails à gauche.

Les prouveurs utilisent déjà cette technique. Les relances consistent à stopper la procédure et à relancer l'analyse avec les informations obtenues à l'analyse précédente, concernant les sous-espaces de recherche qu'il est inutile de considérer.

vers une méthode mixte Il y a dans GSAT de très bonnes bases pour des heuristiques à intégrer à DPLL. Les essais successifs de GSAT peuvent être assimilés aux relances de DPLL. Le choix de la variable dont on va modifier la valeur de vérité est une heuristique sur le choix de l'atome à développer dans *choisi_atome* (utilisée par le prouveur **GRASP**). Une solution intéressante serait donc une procédure Davis-Putnam associée à une recherche locale en priorité.

Mais notons que les heuristiques n'expliquent pas toujours les différences d'efficacité. En effet, les créateurs de **Chaff** [MMZ+01], ont observé que la procédure de propagation *propagation_unitaire* occupe la plus grande partie du temps de calcul (souvent plus de 90%). Ils ont prit le parti de ne pas élaborer d'heuristiques complexes mais de favoriser la rapidité de la propagation.

Chapitre 2

La satisfiabilité en logique modale

En ce qui concerne les logiques non classiques, l'état des recherches sur le problème de satisfiabilité n'est pas aussi fourni qu'en logique classique. Ce n'est qu'à la fin des années 90 que [GS96] propose pour la première fois une technique, l'algorithme KSAT, permettant de tester la validité d'une formule en logique modale dans le système K, au dessus d'une procédure de décision propositionnelle. L'intérêt étant de profiter des avancées sur le problème SAT en logique classique. On parlera plus généralement d'une méthode dite "SAT-based", qui est donc une alternative à celles des tableaux, de traduction fonctionnelle ou du calcul des séquents.

D'abord, nous allons définir le problème de satisfiabilité en logique modale, puis aborderons sa résolution, présentant en détail l'algorithme basique de KSAT. Enfin, nous présenterons ses améliorations possibles ainsi que son extension à d'autres logiques.

2.1 Le problème SAT en logique $K(m)$

Le langage modal $\mathcal{L}_m$ sera celui de la logique propositionnelle classique auquel on ajoute les opérateurs unaires $\Box_1, \Box_2 ... \Box_m$. L'axiomatique est la suivante :

- les tautologies de la logique propositionnelle
- le modus ponens : $\vdash A \rightarrow B$ et $\vdash A \Rightarrow \vdash B$
- le schéma d'axiome K : $\Box_i(A \rightarrow B) \rightarrow (\Box_i A \rightarrow \Box_i B), 1 \leq i \leq m$
- la règle de nécessitation : $\vdash A \Rightarrow \vdash \Box_i A, 1 \leq i \leq m$

Nous redéfinissons $\mathcal{F}$ comme l'ensemble des formules bien formées sur $\mathcal{L}_m$.

$K(m)$-modèle Le $K(m)$-modèle d'une formule ϕ est déterminé par un $(m+2)$-uplet $< W, R_1, ..., R_m, V >$ ayant les propriétés suivantes :

- $W \neq \emptyset$, un ensemble de *mondes possibles*, et soit $w_0 \in W$ le monde *racine*
- $R_i \subseteq W^2, 1 \leq i \leq m$, une relation d'accessibilité entre les mondes possibles, relative à l'opérateur $\Box_i$. On note xR_iy pour indiquer une accessibilité de x à y.
- $V : (\mathcal{F}, W) \rightarrow \{Vrai, Faux\}$
- $V(\alpha \wedge \beta, w) = Vrai \iff V(\alpha) = Vrai$ et $V(\beta) = Vrai$
- $V(\alpha \vee \beta, w) = Vrai \iff V(\alpha) = Vrai$ ou $V(\beta) = Vrai$
- $V(\neg\alpha, w) = Vrai \iff V(\alpha, w) = Faux$
- $V(\Box_i\alpha, w) = Vrai, 1 \leq i \leq m \iff (\forall w', wR_iw' \Rightarrow V(\alpha, w') = Vrai), 1 \leq i \leq m$
- $V(\phi, w_0) = Vrai$

Définition de la $K(m)$-satisfiabilité le problème de satisfiabilité en logique modale $K(m)$ se définit ainsi:

- instance: ϕ une formule modale sur le langage $\mathcal{L}_m$
- question: existe-t-il un $K(m)$-modèle de ϕ?

2.2 Résoudre la satisfiabilité modale

Jusqu'en 1996, les méthodes automatiques de preuve pour les logiques modales étaient essentiellement basées sur les méthodes à tableaux, ou sur la traduction. [GS96] propose une nouvelle technique, qui est développée *au-dessus* des procédures de décision propositionnelle. Nous allons présenter en détail le plus élémentaire des algorithmes SAT-based : KSAT. Mais d'abord, nous introduisons les formalismes nécessaires, ainsi que des propriétés des formules modales du système $K(m)$ (c'est-à-dire avec m modalités de type K), qui justifieront l'algorithme.

2.2.1 Formalisations

On utilise le langage définit en 2.1. Et soit $\mathcal{A} = \{A_1, A_2, ...\}$ un ensemble de propositions primitives.

Définition 11 *On appelle* atome *une formule qui ne peut pas être décomposée propositionnellement (A_k, $\Box_r(A_1 \wedge \neg A_2)$... sont des atomes). Un* littéral *est soit un atome, soit sa négation. Étant donnée une formule ϕ, un atome (ou un littéral) est de $haut - niveau$ pour ϕ s'il n'est dans la portée d'aucun $\Box_i, 1 \leq i \leq m$.*

Définition 12 *Une* affectation de vérité *μ pour une formule modale ϕ est une affectation de vérité à tous les atomes de haut-niveau de ϕ :*

$$\begin{aligned}\mu = \{ \quad & \Box_1\alpha_{11} = Vrai, ..., \Box_1\alpha_{1N_1} = Vrai, \Box_1\beta_{11} = Faux, ..., \Box_1\beta_{1M_1} = Faux, \\ & \Box_2\alpha_{21} = Vrai, ..., \Box_2\alpha_{2N_2} = Vrai, \Box_2\beta_{21} = Faux, ..., \Box_2\beta_{2M_2} = Faux, \\ & ..., \\ & \Box_m\alpha_{m1} = Vrai, ..., \Box_m\alpha_{mN_m} = Vrai, \Box_m\beta_{m1} = Faux, ..., \\ & A_1 = Vrai, ..., A_R = Vrai, A_{R+1} = Faux, ..., A_S = Faux\}\end{aligned}$$

Définition 13 *Une* valeur de vérité r-restreinte *est la restriction de μ à l'ensemble des atomes de la forme $\Box_r\psi, 1 \leq r \leq m$.*

$$\mu^r = \{\Box_r\alpha_{r1} = Vrai, ..., \Box_r\alpha_{rN_r} = Vrai, \Box_r\beta_{r1} = Faux, ..., \Box_r\beta_{rM_r} = Faux\}$$

Définition 14 *Une affectation de vérité μ pour une formule modale ϕ* satisfait propositionnellement *ϕ, noté $\mu \vDash_p \phi$, si et seulement si elle évalue ϕ à $Vrai$, c'est-à-dire, pour tout atome de haut-niveau ϕ_1, ϕ_2 pour ϕ:*

$$\begin{array}{lcl}\mu \vDash_p \phi_1 & \Longleftrightarrow & \phi_1 = Vrai \in \mu \\ \mu \vDash_p \neg\phi_1 & \Longleftrightarrow & \mu \nvDash_p \phi_1 \\ \mu \vDash_p \phi_1 \wedge \phi_2 & \Longleftrightarrow & \mu \vDash_p \phi_1 \text{ et } \mu \vDash_p \phi_2 \\ \mu \vDash_p \phi_1 \vee \phi_2 & \Longleftrightarrow & \mu \vDash_p \phi_1 \text{ ou } \mu \vDash_p \phi_2\end{array}$$

Nous parlerons de μ comme d'un modèle propositionnel d'une formule modale.

2.2.2 l'algorithme KSAT

Voici trois propriétés des formules modales.

Proposition 7 *L'affectation de vérité r-restreinte μ^r est $K(m)$-satisfiable si et seulement si la formule bien formée*

$$\phi_{rj} = \bigwedge_i \alpha_{ri} \wedge \neg\beta_{rj}$$

est $K(m)$-satisfiable, pour chaque $\Box_r\beta_{rj} = Faux$ apparaissant dans μ^r.

Exemple 4 *Soit $\mu^3 = \{\Box_3\psi_0 = Vrai, \Box_3\psi_1 = Faux, \Box_3\psi_2 = Faux\}$. μ^3 sera $K(m)$-satisfiable si et seulement si les formules $\psi_0 \wedge \neg\psi_1$ et $\psi_0 \wedge \psi_2$ sont $K(m)$-satisfiables.*

Proposition 8 *L'affectation de vérité μ est $K(m)$-satisfiable si et seulement l'affectation de vérité r-restreinte μ^r est $K(m)$-satisfiable pour chaque $1 \leq r \leq m$.*

Intuitivement, cela nous autorise à décomposer le problème de recherche de $K(m)$-satisfiabilité en une étape de m problèmes de K-satisfiabilité, et permet aisément de raisonner en logique multi-modale. Si nous parlons d'*une étape*, c'est qu'il est important de voir que l'on ne peut pas remplacer $K(m)$-satisfiabilité par K-satisfiabilité dans la proposition 7. En effet, nous sommes assurés que tous les atomes modaux de haut-niveau sont de la forme $\Box_r\psi_i$, ce qui nous permet de décomposer une étape en simple K-satisfiabilité. Cependant, il se peut que d'autres opérateurs modaux que $\Box_r$ apparaissent dans les ψ_i.

Exemple 5 *Pour que l'affectation de vérité $\mu = \{\Box_0\psi_0 = Vrai, \Box_0\psi_1 = Vrai, \Box_1\psi_2 = Vrai, \Box_0\psi_3 = Faux, \Box_1\psi_4 = Faux, \psi_5 = Vrai, \psi_6 = Faux, \psi_7 = Faux\}$ soit $K(m)$-satisfiable il faudra que les deux affectation de vérités 0-restreinte et 1-restreinte, $\mu^0 = \{\Box_0\psi_0 = Vrai, \Box_0\psi_1 = Vrai, \Box_0\psi_3 = Faux\}$ et $\mu^1 = \{\Box_1\psi_2 = Vrai, \Box_1\psi_4 = Faux\}$ soient $K(m)$-satisfiables.*

Les propositions 7 et 8 permettent de réduire la $K(m)$-satisfiabilité d'une affectation de vérité μ d'une formule à la $K(m)$-satisfiabilité de formules plus petites.

Proposition 9 *Une formule modale ϕ est $K(m)$-satisfiable si et seulement si il existe une affectation de vérité μ $K(m)$-satisfiable telle que $\mu \vDash_p \phi$.*

Cette proposition permet de réduire la $K(m)$-satisfiabilité d'une formule modale ϕ à la $K(m)$-satisfiabilité de ses affectations de vérité μ, modèles propositionnels de ϕ. On voit ainsi apparaître le squelette de l'algorithme :

1. pour une formule ϕ, trouver une affectation de vérité μ pour ϕ telle que μ satisfait *propositionnellement* ϕ ($\mu \vDash_p \phi$).
2. étant données une formule ϕ et une affectation de vérité μ calculée à l'étape précédente. Soient $\Box_r\beta_{rj}$ les atomes modaux de ϕ qui sont évalués à $Faux$ dans μ, et $\Box_r\alpha_{ri}$ ceux qui sont évalués à $Vrai$. ϕ est $K(m)$-satisfiable si pour tout r et j la formule $\phi_{rj} = \bigwedge_i \alpha_{ri} \wedge \neg\beta_{rj}$ est satisfiable.

Notons que la première étape peut être effectuée avec n'importe quelle procédure de décision de la logique propositionnelle classique. C'est aussi par ce raisonnement propositionnel que l'on va tester la satisfiabilité des ϕ_{rj} de la deuxième étape. C'est là que réside tout l'intérêt de KSAT

Mais il est important de voir que si nous utilisons une procédure qui ne permet pas de trouver l'ensemble complet des modèles propositionnels, KSAT ne pourra en aucun cas être complet.

Exemple 6 *Soit $\phi = \Box_1 p \wedge (\Box_1 p \vee \Box_1\neg p \vee \Box_1(p \vee \neg p))$. L'algorithme Davis-Putnam donné en 1.2.1 va trouver le modèle propositionnel $\mu = \{\Box_1 p = Vrai, \Box_1\neg p = Faux, \Box_1(p \vee \neg p) = Faux\}$ et* seulement celui-ci, *qui ne conviendra pas. En effet, par la proposition 7 nous allons nous ramener à la $K(m)$-satisfiabilité de la formule $p \wedge \neg\neg p \wedge \neg(p \vee \neg p)$ qui est inconsistant. Or la formule ϕ est $K(m)$-satisfiable.*

Lorsque l'on a fait le choix de la procédure de décision propositionnelle, il est donc nécessaire d'utiliser la méthode vue en 1.3 pour obtenir un ensemble de modèles complet.

Voici le pseudo-code de l'algorithme KSAT :

$KSAT(\phi)$

Entrée :
ϕ : une formule modale dans un format convenable
pour la fonction $generer\_modele\_propositionnel$
Sortie :
$Vrai$ si ϕ est $K(m)$-consistante, $Faux$ sinon

.....

fonction $KSAT(\phi)$
tant que $\mu = generer\_modele\_propositionnel(\phi)$
si $KSAT_A(\mu)$
alors retourne $Vrai$
$\phi \leftarrow \phi \wedge \neg\mu$
retourne $Faux$

fonction $KSAT_A(\bigwedge_{1\leq r\leq m}(\bigwedge_i \Box_r\alpha_{ri} \bigwedge_j \neg \Box_r \beta_{ri}) \wedge \gamma)$
pour tout $\Box_r$ **faire**
si $KSAT_{AR}(\bigwedge_i \Box_r\alpha_{ri} \wedge \bigwedge_j \neg \Box_r \beta_{rj}) = Faux$
alors retourne $Faux$;
retourne $Vrai$;

fonction $KSAT_{AR}(\bigwedge_i \Box_{r_0}\alpha_{r_0 i} \wedge \bigwedge_j \neg \Box_{r_0} \beta_{r_0 j})$
pour toute conjonction $\neg \Box_{r_0} \beta_{r_0 j}$ **faire**
si $KSAT(\bigwedge_i \alpha_{r_0 i} \wedge \neg\beta_{r_0 j}) = Faux$
alors retourne $Faux$;
retourne $Vrai$;

On suppose que la fonction $generer\_modele\_propositionnel(\phi)$ renvoie un modèle de ϕ lorsque la formule ϕ est consistante, et renvoie $Faux$ dans le cas contraire.

Les procédures $KSAT$, $KSAT_A$ ("A" pour Affectation de vérité) et $KSAT_{AR}$ ("AR" pour Affectation Restreinte) sont respectivement les implémentation directes des propositions 9, 8 et 7, ce qui garantit la correction et la complétude de l'algorithme.

Exemple 7 *Reprenons la formule ϕ de l'exemple 6. On peut identifier $a = \Box_1 p$, $b = \Box_1\neg p$ et $c = \Box_1(p \vee \neg p)$. Nous retombons alors sur la formule propositionnelle de l'exemple 3 dont nous connaissons tous les modèles.*

$generer\_modele\_propositionnel(\phi)$ va donc trouver l'affectation $Aff_0 = \{\Box_1 p = Vrai, \Box_1\neg p = Faux, \Box_1(p \vee \neg p) = Faux\}$. Il n'y a qu'un seul opérateur modal, donc $KSAT_A$ teste Aff_0 en une boucle par $KSAT_{AR}(Aff_0)$. On cherche alors à déterminer la satisfiabilité de la formule $p \wedge \neg\neg p \wedge \neg(p \vee \neg p)$. Or celle-ci est inconsistante et Aff_0 ne convient pas.

Le modèle propositionnel suivant est $Aff_1 = \{\Box_1 p = Vrai, \Box_1\neg p = Faux, \Box_1(p\vee\neg p) = Vrai\}$. Nous vérifions si ce modèle convient en cherchant à déterminer la satisfiabilité de la formule $p \wedge (p \vee \neg p) \wedge \neg\neg p$. L'affectation de p à $Vrai$ convient. Donc l'affectation Aff_1 est une affectation de vérité $K(m)$-satisfiable de la formule ϕ, et donc une preuve de la $K(m)$-satisfiabilité de ϕ.

2.3 Améliorations

Nous venons de voir l'algorithme sous sa forme la plus basique. Les auteurs de [GS96] ont proposé quelques améliorations qui s'avèrent particulièrement simples et efficaces, comme le montrent leurs études expérimentales comparatives.

2.3.1 Trier les atomes modaux

Il est facile de voir que l'algorithme brut tel que nous l'avons présenté, va faire la différence entre deux atomes modaux tels que $\Box_1(\neg A_3 \vee \neg A_1 \vee A_2)$ et $\Box_1(\neg A_1 \vee A_2 \vee \neg A_3)$ qui représentent pourtant la même information. Cela a pour conséquence l'affectation de différentes valeurs de vérité à des formules trivialement équivalentes. Une idée consiste donc à effectuer un pré-traitement sur les formules, en triant les atomes modaux, par un ordre sur les sous-formules.

2.3.2 Factoriser les $\bigwedge_i \alpha_{ri}$

L'observation de la fonction $KSAT_{AR}$ nous permet de voir qu'elle fait plusieurs fois appel à $KSAT$ avec des arguments ayant toujours en commun $\bigwedge_i \alpha_{r_0 i}$. A chaque appel, $KSAT$ essaye de trouver un modèle de la formule $\bigwedge_i \alpha_{r_0 i} \wedge \neg\beta_{r_0 i}$ sans utiliser les calculs effectués précédemment, et en particulier, sans utiliser les affectations de vérité partielles trouvées pour $\bigwedge_i \alpha_{r_0 i}$.

L'idée est de considérer un ensemble $\mathcal{B} = \{\beta_{r_0 1} ... \beta_{r_0 M}\}$ puis chercher des affectations de vérité satisfaisant $\psi = \bigwedge_i \alpha_{r_0 i}$. Chaque fois que l'on en trouve une, on retire de $\mathcal{B}$ les $\beta_{r_0 i}$ qui sont compatibles. On continue ainsi jusqu'à ce que $\mathcal{B}$ soit vide, auquel cas μ^{r_0} est $K(m)$-satisfiable, ou qu'aucun nouveau modèle propositionnel de ψ ne peut être trouvé, ce qui signifie que μ^{r_0} n'est pas $K(m)$-satisfiable.

2.3.3 Vérifier les affectations incomplètes

Une autre amélioration consiste à effectuer des tests de consistance sur les affectations de vérité partielles. En effet, si une affectation partielle est inconsistante, toute extension le sera aussi. Il y a donc fort à gagner à constater ces inconsistances le plus tôt possible.

Il faut pour cela intervenir dans la procédure de décision propositionnelle, c'est-à-dire dans la méthode *generer_modele_propositionnel*. Dans le cas de la procédure DPLL, il s'agit d'insérer un test intermédiaire de $K(m)$-consistance, juste avant les appels récursif.

Cependant, d'un point de vue pratique, nous ne pouvons nous permettre de vérifier chaque affectation partielle, ceci pouvant affecter négativement les performances dans le pire des cas. Il est donc nécessaire de vérifier ces inconsistances à des moments judicieux. Nous introduisons alors une heuristique $probablement\_inconsistant(\mu)$ qui évalue la possibilité que μ puisse être insatisfiable. Voici le pseudo-code illustrant cela, pour une procédure DPLL :

---

generer_modele_propositionnel(ϕ)

Entrée :
ϕ : une formule modale CNF.
Sortie :
Un modèle de ϕ si ϕ est consistante, $Faux$ sinon.

---

fonction *generer_modele_propositionnel*(ϕ)
retourne $KSAT_{DPLL}(\phi, \top)$;

fonction $KSAT_{DPLL}(\phi, \mu)$

$(\phi, \mu) \leftarrow propagation\_unitaire(\phi, \mu)$;

si il existe une clause vide dans ϕ
alors retourne $Faux$;
si il n'y a plus de clause dans ϕ
alors retourne μ
si ϕ ne contient aucune clause positive
soit $\{a_0, ..., a_n\}$ l'ensemble des variables de ϕ;
alors retourne $\mu \bigwedge_{0 \leq i \leq n} \neg a_i$;

si $probablement\_inconsistant(\mu)$
  alors si $KSAT_A(\mu) = Faux$
    alors retourne $Faux$;

$a \leftarrow choisi\_atome(\phi)$;
$KSAT_{DPLL}(\phi \wedge a, \mu)$;
$KSAT_{DPLL}(\phi \wedge \neg a, \mu)$;

Remarquons que dans cette application algorithmique la formule ϕ doit nécessairement être en forme normale conjonctive en raison de l'utilisation de DPLL. L'inconvénient majeur de l'insertion d'un tel test dans la procédure propositionnelle est la perte de son statut de *boîte noire* : nous ne pouvons plus nous contenter d'utiliser une procédure brute. Si toutefois il est décidé de la modifier, il semblerait que l'heuristique $probablement\_inconsistant(\mu)$ pourrait tirer parti des transitions de phases dans la satisfiabilité. Certains travaux ont été menés en ce sens pour la logique $K(m)$ ([HPS, GRS96]) mais ne sont en aucun cas aussi fournis que pour la logique propositionnelle.

2.4 Extension à d'autres logiques

2.4.1 Les logiques normales

L' essentiel de cette section est tiré de [SV98] : les méthodes SAT-based s'obtiennent en utilisant celles par tableaux, en substituant les règles de raisonnement propositionnel de Smullyan ($(\wedge)$, $(\vee)$) par une unique règle, qui est l'application d'une procédure SAT.

Nous présentons d'abord la méthode des tableaux comme dans [Fit83]. Nous verrons ensuite comment il possible de généraliser la méthode dans son traitement propositionnel pour les logiques normales K, KB, K4, D, T, DB, B, D4, S4 et S5. Nous parlerons en général d'une logique $\mathcal{L}$ pour désigner une de ces logiques.

Définition 15 *Une branche de tableau θ est dite* fermée *si elle contient α et $\neg\alpha$ pour une formule α quelconque. Elle est* ouverte *sinon. Un tableau est* fermé *si chacune de ses branches est fermée, et* ouvert *sinon.*

Définition 16 *Un* préfixe *est une suite d'entiers positifs. Un préfixe σ est dit* utilisé *pour une branche d'un tableau θ s'il existe une occurrence de $\sigma : \phi$ sur θ pour une formule ϕ quelconque. Un préfixe σ est dit* non restreint *pour une branche θ d'un tableau si σ ne correspond pas au segment initial d'un autre préfixe utilisé pour θ. Une* extension *d'un préfixe σ, est un préfixe σ' tel que $\sigma' = \sigma, k$ pour un entier k quelconque.*

Un préfixe est un étiquetage d'un monde dans une structure de Kripke. La $\mathcal{L}$-accessibilité est l'accessibilité entre les mondes, fidèle à la propriété sur cette relation d'accessibilité pour la logique $\mathcal{L}$. A savoir : la symétrie pour l'axiome B, la sérialité pour l'axiome D, la réflexivité pour l'axiome T, la transitivité pour l'axiome 4 et l'euclidéanité pour l'axiome 5. Par exemple, la S4-accessibilité sur les préfixes devra satisfaire les conditions de réflexivité et de transitivité.

Un $\mathcal{L}$-tableau se construit comme suit :

- Étape 1 : placer $1 : \phi$ à la racine.
- Étape n+1 : appliquer une des règles suivantes pour la branche θ :

  $(\wedge)\frac{\sigma:\phi_1\wedge\phi_2}{\sigma:\phi_1\,\sigma:\phi_2}$ $\qquad$ $(\vee)\frac{\sigma:\phi_1\vee\phi_2}{\sigma:\phi_1 \mid \sigma:\phi_2}$

  $(\Box)\frac{\sigma:\Box\phi}{\sigma':\phi}$, σ' utilisé pour θ et accessible de σ $\qquad$ $(\neg\Box)\frac{\sigma:\neg\Box\phi}{\sigma':\neg\phi}$, σ' extension non restreinte de σ

[SV98] introduit une fonction f (*générateur de modèles propositionnel*) qui retourne un ensemble complet de modèles propositionnels d'une formule ϕ passée en paramètre (cf. section 1.3) : $f : \mathcal{F} \rightarrow \mathcal{F}^n$. Nous pouvons alors substituer le traitement propositionnel de la méthode des tableaux (règles $(\wedge)$ et $(\vee)$) en une unique règle f.

Définition 17 *Soit $f : \mathcal{F} \rightarrow \mathcal{F}^n$ un générateur de modèles propositionnels, un $\mathcal{L}$-tableau$_f$ est un $\mathcal{L}$-tableau obtenu en substituant les règles propositionnelles ($\wedge$) et ($\vee$) par l'unique règle :*

$$(f)\frac{\sigma : \phi}{\sigma : \mu_1 | ... | \sigma : \mu_n}, \{\mu_1, ..., \mu_n\} = f(\phi).$$

Nous avons alors l'algorithme suivant pour décider la satisfiabilité d'une formule d'une logique $\mathcal{L} \in \{K, KB, D, T, DB, B, D4, S4, S5\}$. La différence de traitement d'une formule en fonction de la logique résidera dans l'accessibilité entre les mondes, caractéristique de la logique considérée, et comme nous l'avons indiqué plus haut dans cette section.

$\mathcal{L} - Tableau_f(\phi)$

Entrée :
ϕ : une formule modale
Sortie :
$Vrai$ si ϕ est $\mathcal{L}$-consistante, $Faux$ sinon

placer $1 : \phi$ à l'origine;
répéter
choisir une occurence d'une formule préfixée $\sigma : \phi$
qui n'est pas terminée, aussi proche de l'origine que possible;
si ϕ n'est pas un littéral propositionnel **alors**
pour toutes les branches θ ouvertes par $\sigma : \phi$ **faire**
si ϕ n'est pas un littéral **alors**
Générer un ensemble complet $f(\phi) = \{\mu_1, ..., \mu_n\}$ de modèles propositionnels de ϕ;
Créer n sous-branches θ_1 ... θ_n pour θ;
Ajouter à θ_i la représentation en formule bien formée de μ_i pour tout i;
sinon si $\phi = \Box\psi$ **alors**
Pour chaque préfixe σ' qui a été utilisé pour θ et qui est $\mathcal{L}$-accessible de σ **faire**
si une aucune occurence de $\sigma' : \psi$ n'est présente dans θ **alors**
Ajouter $\sigma' : \psi$ à θ;
si $\mathcal{L} \in \{D, DB, D4\}$ et il n'existe pas un tel σ' **alors**
Soit k le plus petit entier tel que σ, k est n'est pas restreint pour θ
Ajouter $\sigma, k : \psi$ à θ;
Ajouter une nouvelle occurence de $\sigma : \phi$ à θ;
sinon si $\phi = \neg \Box \psi$ **alors**
soit k le plus petit entier tel que σ, k n'est pas restreint pour θ
Ajouter $\sigma, k : \neg\psi$ à θ;
Déclarer que l'occurence de $\sigma : \phi$ est terminée;
jusqu'à ce que toutes les branches soient fermées
ou que toutes les occurences de formules préfixées soient terminées;
si toutes les branches sont fermées **alors**
retourner $Vrai$;
sinon
retourner $Faux$;

Notons que la terminaison pour des logiques dont l'accessibilité implique la transitivité (K4, S4) est assurée par un test de boucle standard lors de l'application de la règle ($\Box$).

2.4.2 Les logiques non normales

[GTG02] présente une extension de KSAT, dans sa version mono-modale, aux logiques modales classiques E, EM, EN, EMN, EC, EMC, ECN, EMCN.

Soit ϕ une formule de la logique $\mathcal{L}$. On cherche avec un prouveur propositionnel, une affectation de vérité μ aux atomes de haut-niveau de ϕ, qui satisfait la formule propositionnellement. On passe alors μ à $KSAT_{AR}$ pour vérifier sa $\mathcal{L}$-satisfiabilité. Seule ECMN (c'est-à-dire K) est traitée comme nous l'avons vu dans 2.2.2. Les autres sont traitées différemment suivant les particularités

des logiques Le détail de l'aspect algorithmique est donné pour toutes les logiques pré-citées dans [GTG02].

Les méthodes SAT-based de recherche de solutions au problème de satisfiabilité modale sont donc suffisamment robustes pour être utilisées sur des logiques variées. Ces méthodes ont même été les premières à être proposées comme procédures de décision pour les logiques EN, EMN, EC et ECN. Cela renforce l'intérêt de leur étude.

2.5 Discussion

Les procédures de décision SAT-based permettent donc de raisonner sur des logiques modales variées en utilisant au maximum le raisonnement propositionnel. Malgré les travaux de généralisation, l'utilisation de ces méthodes pour des logiques complexes reste un problème délicat. Cependant, des expérimentations sur le comportement de ces méthodes tendent à montrer leur supériorité en complexité computationnelle sur l'ensemble de l'état de l'art. Mais cela reste néanmoins très flou. En effet, de multiples travaux ont clamé successivement la supériorité des méthodes SAT-based puis des méthodes basées sur la traduction en logique du premier ordre introduites par Hans Jürgen Ohlbach et Andreas Herzig. Le lecteur pourra consulter les références [GS96, GGST, HS97, GTG02] où il est question de protocole expérimental, de procédure d'évaluation favorisant l'une ou l'autre des méthodes. Il est néanmoins certain que les méthodes SAT-based dépassent en efficacité les méthodes des tableaux classiques. Les diverses expérimentations ont en effet montré cela et [SM97] en fourni un support théorique.

Partie II

Quelques contributions

Chapitre 3

LoTREC

3.1 Généralités

LoTREC est un démonstrateur de théorèmes développé à l'IRIT au sein de l'équipe LILaC, basé sur la méthode des tableaux, pour les logiques modales et de description (MDL). Son grand atout réside dans sa généricité, mais aussi dans sa convivialité d'affichage des modèles. La plupart des démonstrateurs se limitent à quelques logiques. Or, il existe une infinité de MDL, dont les utilisations dépendent des besoins pour une application particulière. L´état de l'art est donc foisonnant de prouveurs ad hoc, et si un utilisateur souhaite utiliser une logique même peu différente de celles traitées par les systèmes du moment, il devra concevoir son propre démonstrateur. Le problème se pose également lorsque l'on souhaite utiliser des stratégies propres à une application. Un logiciel ne peut pas a priori en proposer un éventail suffisamment large.

LoTREC a été conçu pour combler ce manque. Un utilisateur peut définir des logiques et des stratégies, sans avoir à toucher à l'implémentation du logiciel. Cela représente un grand intérêt dès lors que l'on souhaite effectuer des expérimentations sur différentes logiques et stratégies, ou encore adapter le prouveur à une application.

3.2 Les tableaux "à la toulousaine"

Les travaux menés dans l'équipe LILaC autour de la méthode des tableaux ([CFdCGH97, FdCG02, FDCG01]) sont basés sur l'approche décrite ici.

Définition 18 *Soit $\mathcal{L}$ une logique. Un* $\mathcal{L}$-modèle-graphe *est une structure $\mathcal{G} = (N, R, F)$ telle que:*

- *N est un ensemble de nœuds.*
- *R est une relation binaire sur N.*
- *F est une fonction : $N \rightarrow 2^{\mathcal{F}}$.*

Définition 19 *Un modèle-graphe est dit* fermé *si l'un de ses nœuds contient $\bot$.*

Définition 20 *Soit E un ensemble de formules, un $\mathcal{L}$-tableau pour E est une suite $G^1, ..., G^n, ...$ de modèles-graphes telle que :*

- *$\exists n \geq 1$, G^n est fermé ou $G^n = G^{n+1}$.*
- *G^1 est constitué d'un seul nœud associé à l'ensemble E.*
- *G^{n+1} s'obtient de G^n en appliquant une des règle de $\mathcal{L}$.*

Définition 21 *Un tableau est fermé si il existe un $n \geq 1$ tel que G^n est fermé, il est ouvert sinon. Une formule est fermée par tableau si tous ses tableaux sont fermés, elle est ouverte sinon.*

Les règles classiques Les règles suivantes permettent la réécriture monotone d'une partie de modèle-graphe (uniquement ajout de formules dans un ou plusieurs nœuds et/ou ajout de nœuds et arcs) :

$$
\begin{array}{lll}
(\bot) & \boxed{\sigma, \alpha, \neg\alpha} & \Longrightarrow \quad \boxed{\sigma, \alpha, \neg\alpha, \bot} \\
(\neg\neg) & \boxed{\sigma, \neg\neg\alpha} & \Longrightarrow \quad \boxed{\sigma, \neg\neg\alpha, \alpha} \\
(\wedge) & \boxed{\sigma, (\alpha \wedge \beta)} & \Longrightarrow \quad \boxed{\sigma, (\alpha \wedge \beta), \alpha, \beta} \\
(\vee) & \boxed{\sigma, (\alpha \vee \beta)} & \Longrightarrow \quad \boxed{\sigma, (\alpha \vee \beta), \alpha} \\
 & \text{ou} & \Longrightarrow \quad \boxed{\sigma, (\alpha \vee \beta), \beta}
\end{array}
$$

En raison de la règle $(\vee)$ il y a plusieurs tableaux possibles.

Les règles modales Voici les règles modales pour la logique K.

$$
\begin{array}{lll}
(\neg\,\Box\,\neg) & \boxed{\sigma, \neg\,\Box\,\neg\alpha} & \Longrightarrow \quad \boxed{\sigma, \neg\,\Box\,\neg\alpha} \longrightarrow \boxed{\sigma', \alpha} \\
(K) & \boxed{\sigma, \Box\alpha} \longrightarrow \boxed{\sigma'} & \Longrightarrow \quad \boxed{\sigma, \Box\alpha} \longrightarrow \boxed{\sigma', \alpha}
\end{array}
$$

Théorème 2 ***Adéquation*** *: Si une formule est K-fermée par tableau alors elle est K-insatisfiable.*

Théorème 3 ***Complétude*** *: Si une formule est K-ouverte par tableau (c'est-à-dire que l'un de ses K-tableaux est ouvert) alors elle est K-satisfiable. Le tableau ouvert peut être transformé en un K-modèle.*

3.3 Définition d'une logique

Dans le cadre de **LoTREC**, pour définir une logique, on édite un fichier textuel `.thy`, dans lequel il faut déterminer le langage et donner les règles de tableau. La description du langage se limitera aux connecteurs, les variables et les constantes étant systématiquement à disposition.

Les connecteurs Un connecteur est déterminé par :

- son nom interne.
- le nombre de ses arguments.
- sa propriété d'associativité.
- sa représentation externe.
- sa priorité.

Voici par exemple la définition d'un langage pour une logique modale :

```
connector falsum 0  true  "FALSUM"        4
connector and    2  true  "_ & _"         3
connector not    1  true  "~_"            5
connector nec    1  true  "[]_"           4
```

Les règles de tableau Une règle de tableau décrit comment le tableau évolue. C'est-à-dire, comme un mécanisme d'inférence, donne à partir d'une configuration initiale, les opérations à effectuer sur le tableau. Une règle de tableau se définie donc en deux partie : la description des conditions requises (*descriptor*), et les opérations à appliquer si les conditions sont remplies (*action*). Le détail et l'ensemble des descriptions (*contains*, *hasElement*, *areIdentical* par exemple) et des actions (*link*, *newNode* par exemple) sont donnés dans [Suw01].

Voici, par exemple, les régles de tableau pour la logique K.

```
rule "and"
  descriptor hasElement   node0   (and (variable A) (variable B))
  action     add          node0   (variable A)
  action     add          node0   (variable B)
end

rule "not not"
  descriptor hasElement   node0   (not (not (variable A)))
  action     add          node0   (variable A)
end

rule "not and"
  descriptor hasElement   node0   (not (and (variable A) (variable B)))
  action     duplicate    node0   begin node0 node1 end
  action     add          node0   (not (variable A))
  action     add          node1   (not (variable B))
end

rule "diamond"
  descriptor hasElement   node0   (not (nec (variable A)))
  action     newNode      node0   node1
  action     link         node0   node1 (R)
  action     add          node1   (not (variable A))
end

rule "K"
  descriptor links        node0   node1 (R)
  descriptor hasElement   node0   (nec (variable A))
  action     add          node1   (variable A)
end
```

3.4 Définition d'une stratégie

Après avoir défini une logique, il est nécessaire d'adopter une *stratégie*. Le rôle de la stratégie est de guider la recherche, en déterminant comment combiner et appliquer les règles de tableau. Des stratégies dépendent souvent la complétude et/ou l'adéquation du prouveur, mais aussi les performances.

LoTREC fourni un langage de programmation de stratégies très simple à manipuler. On définit la stratégie dans un fichier textuel `.str`. La syntaxe d'une stratégie est donnée par la grammaire suivante :

liste_de_stratégies ::= stratégie |
stratégie ; liste_de_stratégies
stratégie ::= règle |
repeat stratégie **end** |
allRules liste_de_stratégies **end** |
firstRule liste_de_stratégies **end**

Et on a, en particulier pour la logique K définie précédemment :

règle ::= "and" |
"not not" |
"not and" |
"diamond" |
"K"

- **repeat** S_1 **end** : signifie, répéter la stratégie $S1$.
- **firstRule** $R_1; ...; R_N$ **end** : signifie appliquer la première règle *applicable* parmi les R_i, $1 \leq i \leq N$.
- **allRules** $R_1; ...; R_N$ **end** : signifie appliquer toutes les règles *applicables* parmi les R_i, $1 \leq i \leq N$.

Par exemple, pour la logique K, nous pourrions avoir un fichier `.str` contenant :

```
repeat allRules
                ''and'';
                ''not not'';
                ''not and'';
                ''diamond'';
                ''K''
end
```

qui serait une stratégie naïve, mais qui convient au système K, répétant séquentiellement toutes les règles applicables.

Chapitre 4

ProSaBa : présentation

Cette partie peut être lue comme un manuel de maintenance du logiciel. **ProSaBa** (pour PROlog SAt BAsed) est une implémentation Prolog de la procédure KSAT de [GS96].

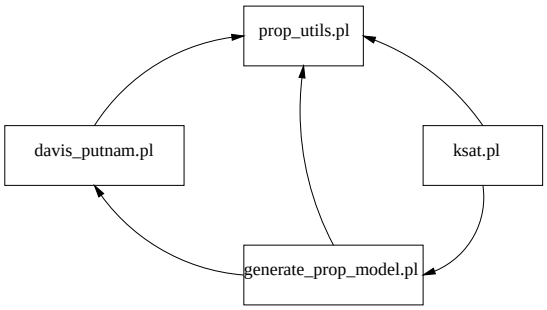

Figure 4.1: Architecture générale de **ProSaBa**. Les flèches dénotent l'utilisation.

Voici le détail des prédicats, classés par *modules* (ou fichiers). Nous ne présentons pas leur signification, mais plutôt la manière de les utiliser. Aussi, nous ne faisons pas apparaître les prédicats qui ne présentent pas d'intérêt particulier, c'est-à-dire, les prédicats qui devraient être considérés comme privés.

prop_utils.pl Les prédicats de `prop_utils` sont des prédicats de manipulation de formules propositionnelles en forme normale conjonctive.

`conjunction(Cnf1,Cnf2,X)`	`X` est la conjonction de `Cnf1` et `Cnf2`
`model2formula(Positive,Negative,X)`	`X` est la conjonction des atomes positifs (`Positive`) avec la conjonction de la négation des atomes négatifs `Negative`
`complement(Set1,Set2,X)`	`X` est le complément de `Set2` dans `Set1`
`negated_atoms(Formula,Positive,X)`	`X` est composé des atomes apparaissant dans `Formula` et qui ne sont pas dans `Positive`
`positive_clause(Clause)`	vrai si `Clause` est une clause positive
`no_positive_clause(Formula)`	vrai si la formule `Formula` ne contient aucune clause positive
`atomslist(Formula,X)`	`X` est la liste des atomes apparaissant dans la formule `Formula`
`cnf_negation(Formula,X)`	`X` est la formule CNF correspondant à la négation de la formule `Formula`

davis_putnam.pl Ce fichier n'est qu'une implémentation de la procédure (complète) de décision propositionnelle que l'on va utiliser pour KSAT. Il n'y a donc un seul prédicat "public" :

`prop_sat(Formula,X)`	`X` est une liste contenant les atomes positifs du modèle trouvé par la procédure

generate_prop_model.pl Ce fichier contient l'implémentation de la méthode permettant de générer un ensemble complet et non redondant de modèles, à partir d'une procédure complète.

`generate_prop_model(Formula,X)`	`X` est une liste contenant les atomes positifs du modèle trouvé par la procédure. Un ensemble complet de modèles sera trouvé.

ksat.pl C'est le module central de **ProSaBa**.

`ksat(Formula)`	vrai si la formule `Formula` est $K(m)$-satisfiable. Les modèles trouvés sont affichés, cf. page 33.
`ktheorem(Formula)`	vrai si la formule `Formula` est valide dans le système K(m).

Remarque sur la couche propositionnelle Il est important de porter une attention particulière à l'indépendance du raisonnement propositionnel et modal afin de tirer au maximum parti des procédures SAT-based.

Même si l'outil permet de traiter de la satisfiabilité des formules modales, la couche propositionnelle est centrale. Il est donc important de pouvoir modifier le prédicat `prop_sat` sans toucher à la partie modale. Cela nous permet en particulier de considérer `davis_putnam.pl` comme une implémentation du prédicat `prop_sat`.

Ainsi, on peut aisément utiliser un prouveur de l'état de l'art. Il suffit pour cela de l'interfacer par l'intermédiaire du prédicat `prop_sat(Formula,X)` en respectant la spécification donnée précédemment.

Chapitre 5

ProSaBa : utilisation

5.1 Installation et exécution

Il faut d'abord copier l'archive prosaba.tar.gz dans un répertoire PROSABA, puis extraire les sources en exécutant les commandes :

```
> gunzip prosaba.tar.gz
> tar xvf prosaba.tar
```

Il reste à compiler les sources. Avec GNU Prolog[1] :

```
> gplc prop_utils.pl davis_putnam.pl generate_prop_model.pl ksat.pl -o proSaBa
> ./proSaBa
```

On peu aussi charger les fichiers dans l'environnement Prolog basique, en entrant aux invites `| ?-` les commandes successives `[prop_utils].`,`[davis_putnam].`,`[generate_prop_model].` et `[ksat].`.

Si l'on souhaite utiliser une autre méthode propositionnelle que celle proposée dans le fichier `davis_putnam.pl`, il faudra fournir à la place un fichier (`prop_sat.pl` par exemple) contenant l'implémentation du prédicat `prop_sat(F,M)`, on créera alors l'environnement **ProSaBa** avec la commande : `gplc prop_utils.pl prop_sat.pl generate_prop_model.pl ksat.pl -o proSaBa`.

5.2 Les formules

La représentation interne d'une formule utilise au maximum la structure de données des listes de Prolog. Il est attendu que la formule soit en forme normale conjonctive.

$\top$	`[]`
$A \vee B$	`[[A,B]]`
$A \wedge B$	`[[A],[B]]`
$\neg a$	`non a`
$\Box_r A$	`r box A`
Exemple : $(a \vee b \vee c) \wedge (\neg a \vee b) \wedge (b \vee \neg c)$	`[[a, b, c], [non a, b], [b, non c]]`
Exemple : $\neg \Box_0 (\neg p \vee q) \vee \neg \Box_0 p \vee \Box_0 q$	`[[non 0 box [[non p,q]], non 0 box [[p]], 0 box [[q]]]]`

5.3 `prop_sat`, `ksat` et `ktheorem`

Les trois commandes principales sont les suivantes :

[1] On peut télécharger GNU Prolog sur http://pauillac.inria.fr/~diaz/gnu-prolog/

`prop_sat` permet de tester la satisfiabilité propositionnelle d'une formule. La commande `prop_sat(Formula,X)` identifie à `X` la liste des atomes positifs d'un modèle de la formule `Formula`, si et seulement si cette formule est satisfiable. On passe au modèle suivant avec le caractère ; et on stoppe l'exécution en enfonçant la touche de retour chariot.

```
| ?- prop_sat([[a, b, c], [non a, b], [b, non c]],X).

X = [c,b] ? ;

X = [b] ?
```

`ksat` permet de tester la K-satisfiabilité d'une formule modale. Lorsqu'un modèle est trouvé, il est affiché (cf. section 5.4). Le déroulement est identique que pour `prop_sat`. Notons que l'on peut utiliser `ksat` pour tester la satisfiabilité d'une formule propositionnelle, auquel cas le modèle complet sera affiché, c'est-à-dire les atomes positifs et négatifs.

```
| ?- ksat([[non 0 box [[non p,q]], non 0 box [[p]], 0 box [[q]]]]).
...
```

`ktheorem` permet de déterminer si une formule modale est valide dans le système K. Si la formule passée en paramètre est un théorème du système K, alors le programme répondra `yes`. Dans le cas contraire, un contre-modèle sera affiché.

```
| ?- ktheorem([[non 0 box [[non p,q]], non 0 box [[p]], 0 box [[q]]]]).

(10 ms) yes
```

5.4 Interpréter les modèles

ProSaBa ne se contente pas de déterminer la satisfiabilité d'une formule, mais affiche aussi les modèles trouvés. Un monde est décrit sur trois lignes :

1. Mod Op : *numéro de la relation d'accessibilité*
2. *clé du monde* from *clés des mondes "pères"* Formula : *formule à satisfaire*
3. + *liste des atomes positifs du modèle propositionnel - liste des atomes négatifs du modèle*

La description du modèle est une suite de descriptions des mondes qui le composent. Aussi, le monde *racine* aura pour père un monde portant la clé `NONE`. Un monde pour lequel on trouvera un modèle propositionnel composé de n atomes négatifs de la forme $\Box_r A$ engendrera n nouveaux mondes. Par exemple, si, pour une formule, un modèle propositionnel est $\{a = \top, b = \bot, \Box_r b = \bot, \Box_r \neg a = \bot\}$, alors le monde engendrera deux nouveaux mondes, par la relation d'accessibilité numéroté r.

Notons que dans la version actuelle de **ProSaBa**, les modèles auront toujours une structure d'arbre, un monde ayant un unique *père*. Pour clarifier les choses, nous présentons deux exemples.

Exemple 8 *Soit* $\phi = \neg\Box_0(\neg\Box_0(\neg p \wedge \neg\Box_0 r)) \wedge (\Box_0 p \vee \Box_0\Box_0\Box_0 r)$. *Elle se traduit pour* ***ProSaBa*** *en*

```
[[non 0 box [[non 0 box [[non p],[non 0 box [[r]]]]]]],
    [0 box [[p]],0 box [[0 box [[0 box [[r]]]]]]]
```

Voici le déroulement de l'exécution :

```
| ?- ksat([[non 0 box [[non 0 box [[non p],[non 0 box [[r]]]]]],[0 box [[p]], 0
 box [[0 box [[0 box [[r]]]]]]]).
Modal Op : 0
1 from 0 Formula : [[0 box [[non p],[non 0 box [[r]]]],[0 box [[0 box [[r]]]]]
+[0 box [[non p],[non 0 box [[r]]]],0 box [[0 box [[r]]]] -[]
Modal Op : 0
2 from 0 Formula : [[non p],[0 box [[0 box [[r]]]]]
+[0 box [[0 box [[r]]]] -[p]
Modal Op : NONE
0 from NONE Formula : [[non 0 box [[non 0 box [[non p],[non 0 box [[r]]]]]],[0
box [[p]],0 box [[0 box [[0 box [[r]]]]]]]
+[0 box [[0 box [[0 box [[r]]]]]] -[0 box [[non 0 box [[non p],[non 0 box [[r]]
]]]],0 box [[p]]]

true ? ;
Modal Op : 0
3 from 0 Formula : [[0 box [[non p],[non 0 box [[r]]]],[p]]
+[0 box [[non p],[non 0 box [[r]]]],p] -[]
Modal Op : 0
6 from 5 Formula : [[non r]]
+[] -[r]
Modal Op : 0
5 from 4 Formula : [[non 0 box [[r]]]]
+[] -[0 box [[r]]]
Modal Op : 0
4 from 0 Formula : [[non 0 box [[0 box [[r]]]],[p]]
+[p] -[0 box [[0 box [[r]]]]]
Modal Op : NONE
0 from NONE Formula : [[non 0 box [[non 0 box [[non p],[non 0 box [[r]]]]]],[0
box [[p]],0 box [[0 box [[0 box [[r]]]]]]]
+[0 box [[p]]] -[0 box [[non 0 box [[non p],[non 0 box [[r]]]]],0 box [[0 box [
[0 box [[r]]]]]]]

true ? ;
Modal Op : 0
7 from 0 Formula : [[0 box [[non p],[non 0 box [[r]]]],[0 box [[0 box [[r]]]],
[p]]
+[0 box [[non p],[non 0 box [[r]]]],0 box [[0 box [[r]]]],p] -[]
Modal Op : NONE
0 from NONE Formula : [[non 0 box [[non 0 box [[non p],[non 0 box [[r]]]]]],[0
box [[p]],0 box [[0 box [[0 box [[r]]]]]]]
+[0 box [[0 box [[0 box [[r]]]]],0 box [[p]]] -[0 box [[non 0 box [[non p],[non
 0 box [[r]]]]]]]

true ? ;

(10 ms) no
```

Par exemple, le second modèle s'interprète comme sur la figure 5.1.

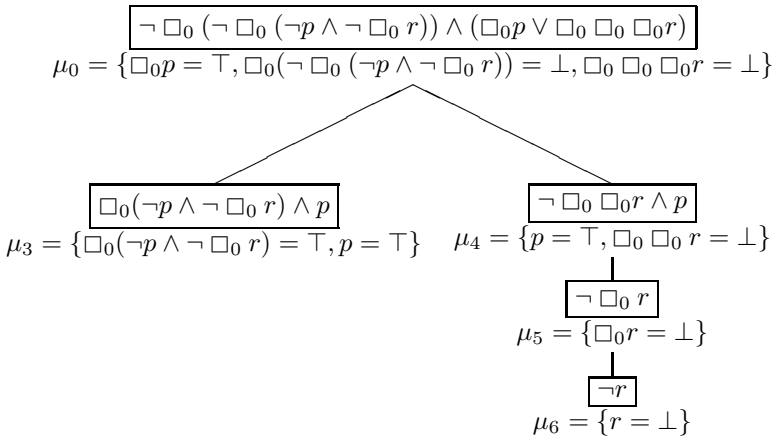

Figure 5.1: Interprétation du second modèle affiché par ***ProSaBa*** *pour la formule* $\phi = \neg\,\Box_0\,(\neg\,\Box_0\,(\neg p \wedge \neg\,\Box_0\,r)) \wedge (\Box_0 p \vee \Box_0\,\Box_0\,\Box_0 r)$. *On accède d'un monde à un autre si une branche les relie et que le monde "père" est représenté au dessus du "fils".*

Exemple 9 *Soit* $\phi' = \Box_0 \neg a \vee \neg\,\Box_0\,(\neg a \vee b) \vee \neg\,\Box_0\,(\neg\,\Box_0\,b \vee \Box_0 \neg a)$.

```
[[0 box [[non a]], non 0 box [[non a,b]],
non 0 box [[non 0 box [[b]], 0 box [[non a]]]]]]
```

Voici le premier modèle trouvé par ***ProSaBa*** *:*

```
Modal Op : 0
1 from 0 Formula : [[a]]
+[a] -[]
Modal Op : 0
2 from 0 Formula : [[a],[non b]]
+[a] -[b]
Modal Op : 0
4 from 3 Formula : [[a],[b]]
+[a,b] -[]
Modal Op : 0
3 from 0 Formula : [[0 box [[b]]],[non 0 box [[non a]]]]
+[0 box [[b]]] -[0 box [[non a]]]
Modal Op : NONE
0 from NONE Formula : [[0 box [[non a]],non 0 box [[non a,b]],non 0 box [[non 0
box [[b]],0 box [[non a]]]]]
+[] -[0 box [[non a]],0 box [[non a,b]],0 box [[non 0 box [[b]],0 box [[non a]]]
]]
```

Il s'interprète ainsi :

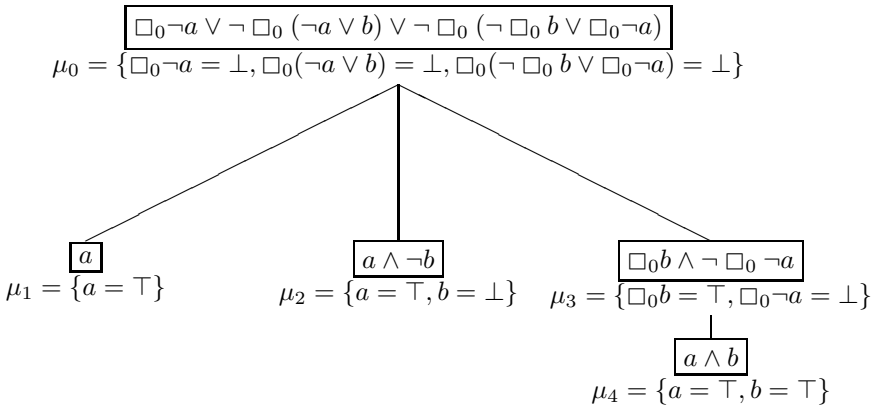

Figure 5.2: Interprétation du second modèle affiché par ***ProSaBa*** *pour la formule* $\phi' = \Box_0\neg a \vee \neg\,\Box_0\,(\neg a \vee b) \vee \neg\,\Box_0\,(\neg\,\Box_0\,b \vee \Box_0\neg a)$*. Les conventions de représentation sont les mêmes que dans l'exemple 5.1.*

Chapitre 6

Perspectives pour LoTREC

Nous avons vu dans les parties 2.4.1 et 2.4.2 que les méthodes SAT-based peuvent être étendues à des logiques variées, mais, malgré l'intéressante généralisation de [SV98] pour un éventail de logiques normales, l'extension à d'autres logiques reste ad hoc. Or comme nous l'avons indiqué dans la section 3.1, la force de **LoTREC** réside dans sa généricité et sa génération de modèles. Il paraît pour le moment exclu de transformer **LoTREC** en un prouveur purement basé sur la satisfiabilité propositionnelle. Cependant, il ressort des méthodes SAT-based une idée intéressante pour les perspectives de **LoTREC** : SAT pourrait être utilisé comme heuristique dans le processus de décision d'une formule modale.

6.1 Utilisation de SAT

Le traitement des connecteurs propositionnels par **LoTREC** est complètement inefficace, or les expérimentations qu'il permet sont surtout concernées par les opérateurs modaux. L'utilisation de SAT permettrait d'améliorer ce point sans nuire à la modularité de **LoTREC** ni à la lisibilité des réfutations qu'il fournit.

L'utilisation d'une règle propositionnelle unique pourrait considérablement améliorer les choses puisque la duplication de tableau aurait lieu seulement lors de l'inspection d'un nouveau modèle propositionnel, ce que l'on voit clairement sur l'application algorithmique page 21.

Lorsque l'on applique une procédure SAT, celle-ci retourne un modèle propositionnel. Nous transformons la $\mathcal{L}$-satisfiabilité d'une formule en une $\mathcal{L}$-satisfiabilité d'un de ses modèles propositionnels. Ce modèle permet de déterminer quelles valeurs booléennes donner aux atomes de haut-niveau. On duplique la branche et, si un atome α est évalué à $Faux$ on ajoute $\neg\alpha$ à la branche et on ajoute α s'il est évalué à $Vrai$. Remarquons que nous sommes certains qu'il n'y aura pas de duplication de branche à l'étape suivante.

Malgré tout, nous pouvons émettre un doute quant à l'efficacité d'une telle méthode. En effet, il est facile de se convaincre que sur certaines instances, le coût de l'application des règles va être plus important avec une règle unique propositionnelle plutôt qu'avec les règles de tableaux traditionnelles. L'exemple suivant en est la preuve :

Exemple 10 *Soit $\phi = \Box p \wedge (\Box p \vee \Box\neg p \vee \Box(p \vee \neg p))$. Un ensemble complet de ses modèles propositionnels a un cardinal de 4 (cf. exemples 3 et 7). Ce qui implique la création de quatre sous-branches. Or, il est facile de voir que pour une méthode des tableaux classique, après l'application de la règle ($\wedge$), la règle ($\vee$) va entraîner la création de seulement trois sous-branches (en supposant une règle ($\vee$) n-aire).*

Mais ces considérations ne mettent pas l'idée d'utiliser SAT comme heuristique complètement à mal. En fait, des cas similaires à l'exemple 10 vont apparaître dans des situations d'instances ayant de nombreux modèles propositionnels, donc des instances sous contraintes et faciles à résoudre en

général (cf. section 1.5.1.1). Nous émettons donc l'hypothèse qu'une telle méthode heuristique peut s'avérer fort efficace sur des instances très contraintes.

Exemple 11 *Soit la formule* $\phi = (a \vee b \vee c) \wedge (\neg a \vee b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$, *dans laquelle on peut identifier les variables* a, b *et* c *à des atomes modaux. Utilisons le prédicat* `generate_prop_model(Formula,X)` *fourni avec* ***ProSaBa*** *pour déterminer un ensemble complet de modèles :*

```
| ?- generate_prop_model([[a,b,c],[non a,b,c],[a,b,non c],[a,non b,non c]],X).

X = [c,b,a] ? ;
X = [a,b] ? ;
X = [c,a] ? ;
X = [b] ? ;

no
```

Dans l'exemple précédent, l'utilisation de SAT comme heuristique va donc entraîner l'utilisation de seulement 4 règles.

Grâce à SAT, nous créons des sous-branches sémantiquement, plutôt que syntaxiquement. Le gain est indéniable par rapport à l'application des règles classiques de tableaux.

6.2 Utilisation de KSAT

Une autre perspective intéressante serait d'aller au-delà de SAT pour guider l'exécution de **LoTREC** et une construction efficace de modèles. En effet, nous avons pensé tirer parti du travail d'une procédure KSAT. L'intérêt réside dans la différence de complexité entre KSAT et une méthode des tableaux classiques, KSAT ayant un meilleur comportement dans le pire des cas. (consulter [SM97] pour les détails).

Proposition 10 *Soit un ensemble d'axiomes* $\mathcal{X} = \{A_1, ..., A_n\}$, $\mathcal{L} \subset \mathcal{X}$ *et* ϕ *une formule modale. Si* ϕ *n'est pas* $\mathcal{L}$*-satisfiable, alors* ϕ *n'est pas* $(\mathcal{L} \bigcup \{A_k\})$*-satisfiable,* $A_k \in \mathcal{X} \setminus \mathcal{L}$.

Cela consiste à déterminer la K-satisfiabilité d'une formule modale en logique $\mathcal{L}$ avec KSAT, qui va indiquer à **LoTREC** la valeur de vérité des atomes de haut-niveau pour obtenir un modèle dans la logique K. Il suffit alors de tester la $\mathcal{L}$-satisfiabilité des K-modèles. **LoTREC** construira dans le même temps un $\mathcal{L}$-modèle pour la formule. Cela, en vertu de la proposition 10.

Nous voyons clairement apparaître qu'un travail semblable peut être réalisé avec une méthode SAT-based pour des logiques plus complexes que K. Par exemple, pour prouver la $S4$-satisfiabilité d'une formule et construire un modèle avec **LoTREC**, nous pourrions utiliser une méthode K4-SAT. Et ainsi de suite. Mais nous ne pouvons a priori déterminer les différences d'efficacité entre des heuristiques SAT, KSAT, K4-SAT... pour **LoTREC**. Seule une expérimentation pourra nous donner des éléments de réponse.

Actuellement, l'architecture de **LoTREC** n'est pas assez flexible pour permettre cela. Lorsqu'elle le sera, nous pourrons tester la collaboration **LoTREC**-**SAT** et **LoTREC**-**ProSaBa**.

Chapitre 7

Discussion et travail futur

L'état de l'art de la logique propositionnelle montre des progrès indéniables qu'il est intéressant d'exploiter pour le raisonnement automatique en logique modale.

Il existe à présent des procédures de décision propositionnelles à haute ingénierie, utilisant des bibliothèques de structures de données particulièrement optimisées pour le maniement de formules logiques, ainsi que des heuristiques efficaces issues des progrès dans la connaissance du problème de satisfiabilité propositionnelle. Les résultats concernant les transitions de phases ou les heavy-tails sont les exemples les plus parlants de ces progrès. Les travaux théoriques sur SAT ont joué un rôle prépondérant dans les approches récentes.

En ce qui concerne les méthodes SAT-based pour raisonner en logique modale, il est dommage qu'il y ait un si grand manque de travaux expérimentaux de comparaison avec les méthodes alternative (tableaux, traduction) pour des logiques moins communes que $K(m)$. Les auteurs de KSAT émettaient pourtant dans [GS96] la conjecture que les techniques SAT-based seraient une réussite plus grande encore, appliquées aux logiques non-normales.

De plus, si la procédure systématique Davis-Putnam a été largement utilisée, les procédures incomplètes telles que GSAT n'ont pas, à notre connaissance, été expérimentées. Mais nous pouvons en effet supposer en considérant les nombreux appels propositionnels des méthodes SAT-based, que la procédure résultante serait fortement incomplète. Nous estimons néanmoins qu'une étude élémentaire serait utile.

Enfin, la méthode utilisée dans la section 2.4.1 pour passer des $\mathcal{L}$-tableaux aux $\mathcal{L}$-tableaux$_f$ suggère une approche plus générale, comme il est dit dans [SV98]. La mise au point d'une adéquation entre méthodes par tableaux et méthodes SAT-based en substituant les règles purement propositionnelles habituelles par une simple règle de génération de modèles propositionnels permettrait d'utiliser l'intégralité de l'état de l'art des tableaux, dont la variété est immense dans la littérature.

Bien qu'il existe des implémentations de KSAT, nous avons développé la nôtre. La motivation de cela est l'intégration à une maquette d'une future version de **LoTREC** en cours de développement en PROLOG. Nous ne doutons pas de la facilité d'une telle mise en place, puisque notre implémentation est extrêmement proche de l'algorithme. Nous avons par ailleurs adapté le générateur d'instances aléatoires créé par les auteurs de ***SAT**. En contre-partie, **ProSaBa** souffre d'une grande inefficacité. Dans une version avancée de **LoTREC** collaborant avec SAT, il sera important d'utiliser un prouveur de l'état de l'art ayant de bons résultats. Chaque année de nouveaux prouveurs libres de droit, toujours plus performants, sont mis au point. Un élément essentiel de l'architecture du futur **LoTREC** sera donc une modification facilitée de la couche propositionnelle.

Il pourra être intéressant de s'inspirer de systèmes existants. Des outils tels que **FACT** [Hor98] et **DLP** [PS] qui bien qu'ils soient présentés comme basés sur la méthode des tableaux, sont des exemples de prouveurs SAT-based, particulièrement efficaces. Notons aussi que la version actuelle de ***SAT** [GTG02] présente une anomalie que nous avons signalé aux auteurs, ne trouvant par

exemple, pas de modèle pour la formule $\Box(p \wedge \neg p)$ dans le système K qui est trivialement satisfiable.

Bibliographie

[AS00] Dimitris Achlioptas and Gregory B. Sorkin. Optimal myopic algorithms for random 3-SAT. In *IEEE Symposium on Foundations of Computer Science*, pages 590–600, 2000.

[CDS+00] Cristian Coarfa, Demitrios D. Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian, and Moshe Vardi. Random 3-SAT: The plot thickens. In *Proceedings of the International Conference on Constraint Programming*, 2000.

[CFdCGH97] M. Castilho, L. Fariñas del Cerro, O. Gasquet, and A. Herzig. Modal tableaux with propagation rules and structural rules. *Fundamenta Informaticae*, 32(3/4), 1997.

[CR92] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science*, 1992.

[DBM00] Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In *Symposium on Discrete Algorithms*, pages 126–127, 2000.

[FDCG01] Luis Fariñas Del Cerro and Olivier Gasquet. Minimal Structures for Modal Tableaux: Some Examples . In *Logic and Logical Philosophy (Parainconsistency. Part II)* , volume 8, pages 1–16. Andrzej Pietruszczak and Jerzy Perzanowski, Department of Logic Nicholas Copernicus University Asnyka 2a 87-100 Torun, 2001. ISBN 83-231-1394-7.

[FdCG02] Luis Fariñas del Cerro and Olivier Gasquet. A General Framework for Pattern-Driven Modal Tableaux . In *Logic Journal of the IGPL* , volume 10, pages 51–83. Dov M. Gabbay, 2002.

[Fit83] Melvin Fitting. *Proof Methods For Modal and Intuitionistic Logics.* D. Reisel Publishing Company, 1983.

[Fri99] E. Friedgut. Sharp threshold of graph properties and the k-sat problem (with an appendix by Jean Bourgain). *J. Amer. Math. Soc.*, 12(4):1017–1054, 1999.

[GGST] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. Sat vs. translation based decision procedures for modal logics: a comparative evaluation.

[GRS96] F. Giunchiglia, M. Roveri, and R. Sebastiani. A new method for testing decision procedures in modal and terminological logics, 1996.

[GS96] Fausto Giunchiglia and Roberto Sebastiani. Building decision procedures for modal logics from propositional decision procedure - the case study of modal k. In *Conference on Automated Deduction*, pages 583–597, 1996.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[GTG02] Enrico Giunchiglia, Armando Tacchella, and Fausto Giunchiglia. Sat-based decision procedures for classical modal logics. *J. Autom. Reason.*, 28(2):143–171, 2002.

[Hor98] Ian Horrocks. The FaCT system. *Lecture Notes in Computer Science*, 1397:307–??, 1998.

[HPS] Ian Horrocks and Peter F. Patel-Schneider. Evaluating optimised decision procedures for propositional modal k (m) satisfiability.

[HS97] Ullrich Hustadt and Renate A. Schmidt. On evaluating decision procedures for modal logic. In *IJCAI (1)*, pages 202–209, 1997.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[PS] Peter F. Patel-Schneider. Dlp system description.

[SBH] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The sat2002 competition.

[SKC93] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[SM97] R. Sebastiani and D. McAllester. New upper bounds for satisfiability in modal logics - the case-study of modal k, 1997.

[SML96] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

[Suw01] S. Suwanmanee. Lotrec. démonstrateur générique de théorèmes. manuel utilisateur et de référence, 2001.

[SV98] Roberto Sebastiani and Adolfo Villafiorita. SAT-Based decision procedures for normal modal logics: A theoretical framework. *Lecture Notes in Computer Science*, 1480:377–388, 1998.

[SW01] John K. Slaney and Toby Walsh. Backbones in optimization and approximation. In *IJCAI*, pages 254–259, 2001.

[Wal99] Toby Walsh. Search in a small world. In *IJCAI*, pages 1172–1177, 1999.

[XL99] Ke Xu and We Li. The sat phase transition, 1999.

[Zha01] Weixiong Zhang. Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *Principles and Practice of Constraint Programming*, pages 153–167, 2001.

Annexes

Annexe A

prop_utils.pl

```
/*****************************************************************
TODO :
- cnf_negation needs simplifications, subsum, trivial clauses elimination
*****************************************************************/

%% non operator
:- op(720,fy,non).                                                    op

                        % Some useful predicates
%% conjunction of two CNF                                             10
conjunction(Cnf_1,Cnf_2,Result_Cnf) :-
        append(Cnf_1,Cnf_2,Result_Cnf).

%% from a model (positive and negative atoms)
%% make a cnf formula (a conjunction of unit clauses)
model2formula([],[],[]) :- !.
model2formula([],[Atom|Negative],[[non Atom]|Formula]) :-
        model2formula([],Negative,Formula),!.
model2formula([Atom|Positive],Negative,[[Atom]|Formula]) :-
        model2formula(Positive,Negative,Formula),!.                   20

%% transform a list of literals into a list of atoms
delnon([],[]).
delnon([non A|X],[A|Y]) :-
        delnon(X,Y),!.
delnon([A|X],[A|Y]) :-
        delnon(X,Y),!.

%% set complement ; a set is a list
complement([],_,[]) :- !.                                             30
complement([A|X],Y,[A|Z]) :-
        \+ member(A,Y),
        complement(X,Y,Z),!.
complement([A|X],Y,Z) :-
        member(A,Y),
        complement(X,Y,Z),!.

%% from a formula and positives atoms of a model
%% finds negatives atoms of the model
negated_atoms(Formula,Positive,Negative) :-                           40
        atomslist(Formula,Atoms),
        complement(Atoms,Positive,Negative).
```

```
                        % POSITIVE_CLAUSE
positive_clause([]).
positive_clause([Literal|Queue]) :-
        \+ Literal = (non _),
        positive_clause(Queue).

no_positive_clause([]).
no_positive_clause([Clause|Queue]) :-
        \+ positive_clause(Clause),
        no_positive_clause(Queue).


                        % list of atoms of a formula

%% list of atoms of a CNF formula
%% an atom may appear more than one time
atomslist_base([],[]) :- !.
atomslist_base([X|Q],Y) :-
        atomslist_base(Q,M),
        delnon(X,L),
        append(L,M,Y),!.

del_doubled([],[]) :- !.
del_doubled([H|Q],R) :-
        member(H,Q),
        del_doubled(Q,R),!.
del_doubled([H|Q],[H|R]) :-
        \+ member(H,Q),
        del_doubled(Q,R),!.

atomslist(F,A) :-
        atomslist_base(F,X),
        del_doubled(X,A).

                        % negation of a CNF

%% negation, [non (non A) ---> A] simplification
neg(non A,A) :- !.
neg(A, non A).

%% from [[a,b],[c,non d]]
%% return [non a,non c], [non a,d], [non b,non c] or [non b,d]
%% (a combinaison of one negated element of each list)
neg_comb([],[]).
neg_comb([Monom|Queue],[Literal|Cnf]) :-
        member(X,Monom),
        neg(X,Literal),
        neg_comb(Queue,Cnf).

%% negation of a CNF
cnf_negation(Formula,Cnf) :-
        findall(X,neg_comb(Formula,X),Cnf).
```

Annexe B

davis_putnam.pl

```
/*************************************************************
representation of the formulae in CNF :
(A \/ ~B \/ C) /\ (~A \/ D)
is translated into [[A,non B, C],[non A, D]]

TODO :
- improve heuristic choose_atom
*************************************************************/

%% non operator                                                    10
:- op(720,fy,non).                                                 op

%% interface
prop_sat(X,Y) :-
        davis_putnam(X,Y).

                                % DAVIS PUTNAM
davis_putnam([],[]).
davis_putnam(Formula,_) :-
        member([],Formula),                                        20
        fail.

%% Unit Propagation
davis_putnam(Formula,[Atom|Model]) :-
        member([Atom],Formula),
        \+ Atom = (non _),
        assign(Atom,true,Formula,X),
        davis_putnam(X,Model),!.
davis_putnam(Formula,Model) :-
        member([non Atom],Formula),                                30
        assign(Atom,false,Formula,X),
        davis_putnam(X,Model),!.

%% no positive clause
davis_putnam(Formula,[]) :-
        no_positive_clause(Formula),!.

%% Splitting
davis_putnam(Formula,Model) :-
        choose_atom(Formula,Atom),                                 40
        ((
          assign(Atom,true,Formula,X),
```

```
        [Atom|M] = Model,
        davis_putnam(X,M)
      );
      (
        assign(Atom,false,Formula,X),
        M = Model,
        davis_putnam(X,M)
      )).

                        % CHOOSE_ATOM

%% an atom in a smallest postive (TODO) clause
choose_literal(Formula,Literal) :-
      choose_literal(Formula,Literal,_).
choose_literal([Clause|[]],Literal,Size) :-
      last(Clause,Literal),
      length(Clause,Size),!.
choose_literal([Clause|Queue],Literal,Size) :-
      choose_literal(Queue,Literal,Size),
      length(Clause,L),
      Size =< L,!.
choose_literal([Clause|Queue],Literal,Size) :-
      choose_literal(Queue,_,L),
      length(Clause,Size),
      Size < L,
      last(Clause,Literal),!.

choose_atom(F,A) :-
      choose_literal(F,X),
      (X = (non A) ; (\+ X = (non _), X = A)).

                        % PROPAGATION (ASSIGN)
assign(_,_,[],[]).

%% Subsumption
assign(Atom,true,[Clause|Queue],Result) :-
      member(Atom,Clause),
      assign(Atom,true,Queue,Result),!.
%% Resolution
assign(Atom,true,[Clause|Queue],[X|Result]) :-
      member(non Atom,Clause),
      delete(Clause,non Atom,X),
      assign(Atom,true,Queue,Result),!.
assign(Atom,true,[Clause|Queue],[Clause|Result]) :-
      !,assign(Atom,true,Queue,Result).

assign(Atom,false,[Clause|Queue],Result) :-
      member(non Atom,Clause),
      assign(Atom,false,Queue,Result),!.
assign(Atom,false,[Clause|Queue],[X|Result]) :-
      member(Atom,Clause),
      delete(Clause,Atom,X),
      assign(Atom,false,Queue,Result),!.
assign(Atom,false,[Clause|Queue],[Clause|Result]) :-
      assign(Atom,false,Queue,Result),!.
```

Annexe C

generate_prop_model.pl

```
/*************************************************************
*************************************************************/

%% to generate a complete set of models of a propositional formula
%% can be used with any complete SAT procedure prop_sat
generate_prop_model(Formula,Model) :-                                    generate_prop_model
        prop_sat(Formula,Model_1),!,
        (
          Model_1 = Model;
          (                                                              10
            negated_atoms(Formula,Model_1,Negative),
            model2formula(Model_1,Negative,X),
            cnf_negation(X,NegMod1),
            conjunction(Formula,NegMod1,Fprime),
            generate_prop_model(Fprime,Model)
          )
        ).
```

Annexe D

ksat.pl

```
/*************************************************************
NOTE :
- representation of a formula must be in a format accepted by the
generate_prop_model procedure.
- Nec(0)(a\/b) is translated into [[0 box [[a,b]]]].

EXAMPLES :
ksat([[non 0 box [[non p,q]], non 0 box [[p]], 0 box [[q]]]]).     (AXIOM K)
ksat([[0 box [[non p,q]]],[0 box [[p]]],[non 0 box [[q]]]]).       (NEG K)
ktheorem([[non 0 box [[non p,q]], non 0 box [[p]], 0 box [[q]]],                10
          [non 1 box [[non p,q]], non 1 box [[p]], 1 box [[q]]]]). (AXIOM K(2))

ksat([[0 box [[non a]], non 0 box [[non a,b]],
       non 0 box [[non 0 box [[b]], 0 box [[non a]]]]]]).

ksat([[non 0 box [[non 0 box [[non p],[non 0 box [[r]]]]]]],
      [0 box [[p]], 0 box [[0 box [[0 box [[r]]]]]]]]).

TODO :
- improvements (sorting modal atoms,                                           20
                factorizing positives conjunctions,
                check for incomplete assignments)
*************************************************************/

%% box operator
:- op(720,xfy,box).                                                           op

%% initialise dynamics clauses
init_dynamics :-
        asserta(counter(0)).                                                  30

%% a key is a unique identifiant for a world of a model
next_key(X) :-
        retract(counter(Y)),
        X is Y+1,
        asserta(counter(X)).

%% next_level_formula. transform a model in a cnf composed
%% of the formulae under the scope of a R-box
next_level_formula(_,[],[]) :- !.                                             40
next_level_formula(R,[R box A|X],Z) :-
        conjunction(A,Y,Z),
```

```
        next_level_formula(R,X,Y),!.
next_level_formula(R,[_|X],Y) :-
        next_level_formula(R,X,Y),!.

%% keep in the second list, only R-modal atoms of the first list
keep_box(_,[],[]) :- !.
keep_box(R,[R box X | Atoms],[R box X | Result]) :-
        keep_box(R,Atoms,Result),!.                                    50
keep_box(R,[_|Atoms],Result) :-
        keep_box(R,Atoms,Result),!.

%% list box operators of the first list in the second one
list_mod_ops([],[]) :- !.
list_mod_ops([R box _ | Cdr],[R|Boxes]) :-
        list_mod_ops(Cdr,Boxes),
        \+ member(R,Boxes),!.
list_mod_ops([_|Cdr],Boxes) :-
        list_mod_ops(Cdr,Boxes).                                       60

%% print a model
print_model(Formula,Positive,Negative,PreviousKey,Key,ModOp) :-
        write('Modal Op : '), write(ModOp), nl,
        write(Key), write(' from '), write(PreviousKey),
        write(' Formula : '), write(Formula), nl,
        write('+'), write(Positive), write(' '),
        write('-'), write(Negative), nl.

                                                                       70
                        % KSAT

%% test if a formula is satisfiable in system K(m)
ksat(Formula) :-
        init_dynamics,
        ksat(Formula,_,'NONE',0,'NONE').
ksat([],_,_,_) :- !. % is this clause useful ?
ksat(Formula,Positive,PreviousKey,Key,ModOp) :-
        generate_prop_model(Formula,Positive),
        negated_atoms(Formula,Positive,Negative),                      80
        append(Negative,Positive,NegPos),
        list_mod_ops(NegPos,ListModOps),
        ksat_assignment(ListModOps,Positive,Negative,PreviousKey,Key),
        print_model(Formula,Positive,Negative,PreviousKey,Key,ModOp).

ksat_assignment([],_,_,_,_) :- !.
ksat_assignment([R|List],Positive,Negative,PreviousKey,Key) :-
        keep_box(R,Positive,PosR),
        keep_box(R,Negative,NegR),
        ksat_restricted_assignment(R,PosR,NegR,PreviousKey,Key),       90
        ksat_assignment(List,Positive,Negative,PreviousKey,Key).

ksat_restricted_assignment(_,_,[],_,_) :- !.
ksat_restricted_assignment(R,Positive,[R box B|Beta],PreviousKey,Key) :-
        next_level_formula(R,Positive,Cnf),
        cnf_negation(B,Neg),
        conjunction(Neg,Cnf,Formula),
        next_key(NextKey),!,
        ksat(Formula,_,Key,NextKey,R),
        ksat_restricted_assignment(R,Positive,Beta,PreviousKey,Key).   100
```

```
%% test if a formula is a theorem in system K(m)
ktheorem(X) :-
        cnf_negation(X,Neg),
        \+ ksat(Neg).
```