FACULTÉ DES SCIENCES ET TECHNIQUES
UNIVERSITÉ DE BOURGOGNE

THÈSE

$\mathcal{SACRE}$: une Approche de Résolution en Logique fondée sur des techniques de Satisfaction de Contraintes

présentée par

Jean-Michel RICHER

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE BOURGOGNE

(Spécialité Informatique)

thèse soutenue le Mardi 5 Janvier 1999 devant le jury composé de

Rapporteurs	Mr Bruno LEGEARD
	Mr Antoine RAUZY
Examinateurs	Mme Jacqueline CHABRIER
	Mr Jean-Jacques CHABRIER
	Mr Michel RUEHER
	Mr Laurent TRILLING

Remerciements

Cette thèse est le fruit d'un travail personnel dans lequel je me suis investi depuis bientôt trois ans. Elle représente pour moi l'achèvement de ma formation universitaire et je suis heureux d'avoir ici l'occasion de remercier tous mes professeurs, parents et amis proches qui m'ont formé, influencé et encouragé.

Je remercie tout d'abord Bruno LEGEARD et Antoine RAUZY qui m'ont fait l'honneur de rapporter ce travail et dont les remarques m'ont permis de finaliser cette thèse.

Je suis également reconnaissant à Michel RUEHER d'avoir bien voulu accepter de participer à mon jury de thèse ainsi que Laurent TRILLING qui fut élu président de ce jury.

Je remercie Jacqueline CHABRIER qui fut ma responsable en D.E.A et qui a également accepté de participer à ce jury.

Ma gratitude va également à mon directeur, Jean-Jacques CHABRIER qui m'a contraint *sans cesse à remettre sur le métier mon ouvrage* afin d'aboutir à un résultat impeccable et professionnel. Ses conseils éclairés, ses lectures et son intérêt pour ma recherche m'ont permis de mener à terme ce travail.

Je tiens à remercier toute l'équipe du LIRSIA où j'ai pu travailler dans une ambiance chaleureuse et amicale grâce à François JACQUENET, Irène FOUCHEROT, Dominique BELIME et Loutfi SOUFI.

J'adresse également mes remerciements aux *anciens* du LIRSIA avec qui j'ai pu avoir un contact privilégié. Merci à Marc BERNARD, Vincent JULIARD, Claire NICOLINI et Isabelle THIBAUT qui eut le courage de relire et corriger certaines parties de cette thèse.

Enfin, un grand merci à Donald E. KNUTH et Leslie LAMPORT qui ont élaboré respectivement TEX et LATEX, logiciels sans lesquels cette thèse aurait difficilement vu le jour.

Je dédie cette thèse à mes parents sans le soutien desquels je n'aurais pu accomplir ce travail

Il y a celui qui ne sait pas,
Celui qui croit savoir
Et celui qui sait.

Celui qui ne sait pas
Apprend des autres.
Celui qui croit savoir
Enseigne celui qui ne sait pas
Et celui qui sait...

Celui qui sait se tait.

Table des matières

Liste des tableaux

Table des figures

Introduction

Les mécanismes de raisonnement chez l'homme sont encore aujourd'hui sources de nombreuses interrogations. La preuve de théorèmes, qui est à la base des mathématiques, est une activité de l'esprit humain qui consiste à déduire (ou inférer) des conclusions qui découlent selon des principes *valides* d'un ensemble de données premières (ou axiomes). L'une des caractéristiques de l'esprit humain réside dans sa capacité à trouver des preuves rapidement en se focalisant sur un nombre restreint de points de choix, en éliminant d'office les chemins considérés comme infructueux. Ce mode de pensée *heuristique* se révèle moins fiable qu'une recherche exhaustive, mais néanmoins plus efficace.

En revanche, les ordinateurs modernes ne disposent pas de cette faculté mais offrent de nouveaux procédés d'investigation de la preuve de théorèmes en raison de leur puissance de calcul et de leur précision.

La **Démonstration Automatique** s'intéresse à la preuve de théorèmes dans le cadre de la logique. Un démonstrateur automatique de théorèmes est un programme qui utilise des procédés de déduction, appelés *règles d'inférence*, pour inférer des conclusions qui découlent logiquement d'un ensemble d'axiomes. Bien que les ordinateurs soient capables de réaliser des inférences à une vitesse vertigineuse, beaucoup plus rapidement qu'aucun cerveau humain, il n'en reste pas moins que la rapidité ne représente pas toujours la panacée. La recherche de preuve en logique reste un problème combinatoire complexe car elle s'attaque à des problèmes NP-complets. La complétude des méthodes employées pour la recherche de preuve implique de parcourir l'espace de recherche dans sa totalité. On est donc parfois amené à s'intéresser à des branches de l'arbre de recherche stériles pour le but à résoudre. La solution qui consiste à vouloir tirer parti de la force brute des machines en cherchant à produire toutes les inférences possibles se révèle souvent une solution désastreuse, hors de portée de la puissance des calculateurs.

La nécessité de disposer de *mécanismes de contrôle aptes à orienter la recherche* reste à l'heure actuelle une préoccupation légitime des chercheurs. Ces mécanismes sont qualifiés de *stratégies de recherche*. En conséquence, l'efficacité d'un système de démonstration automatique repose sur la conception de règles d'inférence plus performantes alliées à des stratégies de recherche plus fiables ainsi qu'à des heuristiques performantes.

La **Programmation en Logique** représente une spécialisation de la Démonstration Automatique qui privilégie l'efficacité des traitements en se restreignant à des clauses de *Horn* qui capturent le caractère procédural de certaines clauses. Un programme logique représente l'expression logique d'un algorithme qui possède l'avantage de pouvoir orienter la recherche dans une direction précise limitant ainsi l'étendue de l'espace de recherche à étudier. Cependant, les clauses de *Horn* seules sont insuffisantes pour spécifier de nombreux problèmes. En particulier, il est impossible de formuler des informations négatives. Un langage comme *Prolog* introduit un mécanisme spécifique (prédicat not) pour traiter une forme de négation (négation par l'échec, basée sur l'hypothèse du monde clos). Théoriquement, on devrait être en mesure de proposer les deux formes de négation (logique et par l'échec). L'utilisation de prédicats "*extra-logiques*", comme la coupure (cut), augmente l'efficacité des traitements mais pénalise la déclarativité et influe sur la complétude de la méthode de résolution. Par ailleurs, se pose le délicat problème de la représentation des connaissances. Le rôle d'un langage de représentation de connaissances en logique est de permettre une description aussi naturelle que possible des problèmes à traiter. Idéalement, un tel langage devrait être *déclaratif* et *statique*. Il devrait également offrir la possibilité d'exprimer des informations négatives ainsi que des informations incomplètes, ce qui n'est pas le cas de *Prolog*. Le formalisme des clauses sous forme normale conjonctive (CNF) répond à ces exigences.

Dans cette thèse, nous nous intéressons à la conception d'une nouvelle méthode de résolution des problèmes de la logique des prédicats du premier ordre exprimés sous forme normale conjonctive. L'originalité de notre approche repose sur la résolution des problèmes de la logique en tant que **Problèmes de Satisfaction de Contraintes** (CSP) adaptés au calcul des prédicats. Ces CSP peuvent être résolus comme des programmes logiques et secondés par des techniques heuristiques performantes qui visent à limiter l'espace de recherche à étudier. La méthode qui résulte de nos travaux, baptisée *Sacre*, allie la déclarativité de la Démonstration Automatique à l'efficacité de la Programmation en Logique et de la Programmation par Contraintes. Elle permet de réaliser du *test de consistance* et du *calcul de modèle* sur des problèmes exprimés dans un langage de description de connaissances *déclaratif* et *statique*.

Motivations

Dans ce chapitre, nous introduisons de manière très générale les différents concepts qui ont motivé notre approche. Nous montrons que Démonstration Automatique, Programmation en Logique et Programmation par Contraintes possèdent des similitudes qui favorisent leur rapprochement.

1 Démonstration Automatique

Le rôle d'une théorie est de procurer une vision explicative d'un certain nombre de phénomènes concernant un domaine d'étude donné. Une théorie est formée par un ensemble de propositions ou vérités premières et un ensemble de règles de déductions qui permettent de déduire de nouvelles propositions, considérées comme des vérités de la théorie. Poursuivre une expérience dans le cadre d'une théorie consiste à établir la relation de causalité qui lie les conditions initiales de l'expérience, appelées hypothèses, à l'ensemble de conditions finales, appelées conclusions. L'essence d'une théorie est de pouvoir prédire de manière systématique les conclusions dérivant d'un ensemble d'hypothèses. On peut alors utiliser une théorie de deux façons différentes : soit en tant qu'*outil de validation* afin de vérifier que les conclusions d'une expérience et celles obtenues par la théorie concordent ; soit en tant qu'*outil de prédiction* dans le cas où réaliser une expérience n'est pas possible.

L'activité de raisonnement est le procédé par lequel, à partir d'une connaissance initiale, on parvient à exhiber d'autres connaissances (dites émergentes) qui ne sont pas exprimées de façon explicite dans la connaissance initiale et qui découlent de celle-ci suivant une relation de causalité. Les phénomènes physiques sont des phénomènes complexes bien souvent entourés d'un ensemble d'artefacts qui font obstacle à la connaissance *vraie*. La théorie représente *une tentative d'épuration de ce qui entrave notre compréhension de la réalité* ainsi qu'un procédé de raisonnement fiable.

L'objet du **Raisonnement Automatique** est, selon Larry **Wos** ([132] p. 5), de concevoir des programmes qui procurent une assistance lors de la résolution de problèmes faisant appel à des procédés de raisonnement. Il existe de nombreux paradigmes de représentation des connaissances (réseaux de neurones, réseaux sémantiques, Frame, etc ...), parmi lesquels la logique, et en particulier le calcul des prédicats, fournit un cadre formel rigoureux, largement étudié depuis l'Antiquité et facilement implantable sur un ordinateur. La **Démonstration Automatique**[1] (DA) est une branche du Raisonnement Automatique qui s'est fixée pour objectif la résolution par des mécanismes d'inférence de problèmes exprimés en logique. Le spectre des problèmes étudiés dans le cadre de la DA s'étend du simple problème de réflexion, aux preuves de théorèmes mathématiques, en passant par la conception et la vérification de circuits logiques.

Bien que les origines de la Démonstration Automatique, telle que nous la connaissons actuellement, remontent aux débuts de l'Informatique, celle-ci a su en moins d'un demi-siècle évoluer jusqu'au stade d'une discipline aujourd'hui largement reconnue et appréciée. Par exemple, dans le domaine des Mathématiques, les logiciels de DA fournissent un excellent outil d'étude de nombreux problèmes. Ainsi, a-t-on pu répondre, ces dernières années, à un certain nombre de questions qui étaient jusque-là restées ouvertes, grâce à des logiciels[2] de DA. En particulier, on a pu déterminer en 1982 [127] que dans le cadre des algèbres booléennes ternaires, inventées en 1943, chacun des trois premiers axiomes était indépendant

1. ou Démonstration Automatique de Théorèmes
2. ou démonstrateurs de théorèmes

du quatrième. Il convient cependant de noter que la résolution d'un tel problème ne peut se faire sans l'aide d'un opérateur humain capable d'orienter la recherche.

Une grande variété de mécanismes de résolution pour la Démonstration Automatique ont été proposés en fonction de différents critères de conception. D'un point de vue très général, on peut classer ces mécanismes en deux catégories :

– la première concerne la *recherche de consistance* (ou *recherche de preuve*) et a trait à l'aspect syntaxique de la logique. Parmi ces méthodes on trouve les systèmes de Gentzen [49], les systèmes de Hilbert, la Déduction Naturelle [95], les Tableaux sémantiques [114] pour les formes normales disjonctives et le *principe de résolution* [104] pour les formes normales conjonctives,
– la seconde, qualifiée de *recherche de satisfiabilité*[3] (ou *calcul de modèle*), touche à l'aspect sémantique de la logique. Elle consiste à déterminer un modèle ou un contre-exemple d'un problème. On peut également, dans cette catégorie, marquer une distinction entre les approches basées sur la *réduction* ou l'*extension*. Les approches fondées sur la réduction génèrent de manière itérative des instanciations d'un problème et en testent la satisfiabilité. Les approches basées sur l'extension tentent de construire un modèle en faisant l'hypothèse de la valeur de vérité de certains faits.

D'un point de vue théorique, le rôle d'un démonstrateur automatique est de faire de la *preuve de théorème*, c'est-à-dire de démontrer qu'une formule F découle logiquement d'un ensemble d'axiomes H (ce que l'on note $H \vdash F$). D'après le théorème de déduction formelle (voir $1^{ère}$ Partie, section 1.2.1, théorème 1.2.1), montrer que F est conséquence logique de H est équivalent à montrer que la formule logique $H \rightarrow F$ est un théorème dérivant des axiomes de la logique, ou encore, d'après l'équivalence syntaxe/sémantique, que $H \rightarrow F$ est une formule valide. On peut démontrer cela de deux manières différentes : soit par l'*affirmation*, en montrant effectivement que $H \rightarrow F$ est valide, soit par la *réfutation* (mode de raisonnement analogue au raisonnement par l'absurde), en montrant que $H \wedge \neg F$ est insatisfiable. Dans ce dernier cas, la transformation de la formule $H \wedge \neg F$ sous forme de *clauses* fournit une procédé de preuve intéressant car il permet de travailler sur une formule plus simple que la formule initiale, non logiquement équivalente à celle-ci, ce qui n'a aucune incidence sur le résultat final étant donné que les clauses se contentent de retenir le caractère de satisfiabilité de la formule initiale. Un autre aspect du Raisonnement Automatique concerne la *déduction*, soit la production des théorèmes qui découlent logiquement d'un ensemble d'axiomes. La distinction entre preuve de théorème d'une part, et déduction d'autre part, n'est généralement pas soulignée car ces deux aspects sont assez proches. L'intérêt de la preuve de théorème par rapport à la déduction tient à la terminaison de la résolution si le théorème à démontrer est conséquence logique des axiomes.

Les premiers résultats fondamentaux vers la mécanisation de la Démonstration Automatique furent établis par **Herbrand** [55]. Jacques Herbrand propose en 1930 un procédé de résolution des problèmes exprimés en logique des prédicats, basé sur la recherche de modèles. Sa méthode, bien qu'attrayante se révèle difficile à conduire manuellement. L'invention de l'ordinateur suscita un regain d'intérêt pour la DA et l'une des premières approches fondée sur les travaux de Herbrand fut celle de **Gilmore** [50] en 1960. Cette approche basée sur la réduction se révéla inefficace en raison de la multitude d'interprétations résultant de la génération de la base de Herbrand. La découverte du principe de résolution par **Robinson** [104] en 1965, ouvre la voie à une nouvelle génération de démonstrateurs de théorèmes beaucoup plus efficaces.

Cependant, ainsi que nous l'avons mentionné dans l'introduction de cette thèse, la Démonstration Automatique de Théorèmes s'attaque à la résolution de problèmes NP-complets et n'a que peu de chance d'aboutir si elle se base uniquement sur la puissance de calcul des ordinateurs.

On pourraît effectivement penser que les évolutions technologiques que subissent les composants électroniques, augmentant leurs performances à une allure incroyable, se répercutent sur l'efficacité des logiciels. Malheureusement, les problèmes traités en Démonstration Automatique sont très complexes, beaucoup trop complexes en tout cas pour que l'augmentation de la vitesse des processeurs soit un facteur décisif de leur résolution ; la taille de l'espace de recherche de certains problèmes étant parfois suffisante à "*décourager*" le plus performant des cerveaux électroniques. La solution semble donc résider dans la combinaison entre vitesse de calcul des processeurs et l'utilisation de procédés stratégiques et heuristiques

3. on dit également satisfaisabilité.

qui orientent la recherche dans le sens d'une solution ou qui élaguent les parties infructueuses de l'arbre de recherche. La recherche de tels procédés ainsi que la conception de mécanismes d'inférence toujours plus puissants représente un domaine d'études d'un intérêt constant.

Il semble difficile de dégager les tendances actuelles en matière de Démonstration Automatique. Cependant, on a récemment assisté à une recrudescence de l'intérêt porté aux méthodes sémantiques seules [111, 135] ou couplées à des méthodes syntaxiques [18, 24, 112, 70]. Il est largement reconnu que l'approche syntaxique seule n'est pas assez puissante pour concevoir des méthodes de preuves efficaces. La connaissance du domaine de discours du problème est fondamentale. On n'abordera pas en effet la résolution d'un problème de géométrie de la même manière que celle d'un problème d'algèbre. On peut par exemple coder la connaissance du problème sous forme de modèles ou d'exemples qui seront utilisés pour guider la recherche [12, 100].

2 Programmation en Logique

La **Programmation en Logique** [68] est née de l'idée que la logique pouvait être utilisée en tant que langage de programmation. Une des notions clé de la Programmation en Logique, due à **Kowalski** [62], est l'idée selon laquelle un algorithme consiste en deux composantes disjointes : *logique* et *contrôle*.

La distinction fondamentale entre Programmation Impérative et Programmation en Logique réside dans la spécification des composantes *logique* et *contrôle*. La composante *logique* d'un problème caractérise le « *quoi* », c'est-à-dire, ce qui doit être résolu, alors que la composante *contrôle* caractérise le « *comment* », ou comment résoudre le problème. En programmation impérative, on spécifie les deux composantes que sont *logique* et *contrôle* (voir figure 1) :

```
function factorielle( n : integer ) : integer ;
var i, f : integer ;
begin
    f := 1 ;
    for i := 1 to n do f := f * i ;
    factorielle := f ;
end ;
```

FIG. 1 – *Programmation impérative en Pascal*

En Programmation en Logique, par contre, un programme décrit le résultat auquel on désire aboutir : le programmeur spécifie le *quoi* sans se soucier du *comment*. On parle alors de programmation *déclarative* (cf. fig. 2). Le contrôle est quant à lui laissé au système de résolution de problèmes.

```
factorielle(0,1).
factorielle(N,M) :- N1 is N-1,
                    factorielle(N1,M1),
                    M is N*M1.
```

FIG. 2 – *Programmation déclarative en Prolog*

Cette déclarativité est cependant relative car l'exécution d'un programme logique est influencée par l'ordonnancement des clauses de programme et des littéraux au sein même des clauses, comme par exemple en *Prolog*. Le langage *Prolog*[4] [28, 27] est fondé sur une implantation du principe de résolution SLD[5] [3]. La résolution SLD est un raffinement complet et correct [68] du principe de résolution dont l'objectif est de fournir une réponse calculée à un but. Néanmoins, la stratégie de recherche utilisée par *Prolog* est une stratégie *en profondeur d'abord* avec retour arrière et sélection séquentielle des clauses, entraînant ainsi la perte de complétude de la méthode, notamment dans le cas de branches infinies. Par exemple, si on écrit

4. acronyme de PROgrammation en LOGique.
5. Linear resolution with Selection function for Definite clauses.

le programme de la factorielle en intervertissant les deux clauses qui le composent, l'interpréteur *Prolog* entre dans une boucle infinie.

L'exemple précédent montre que *Prolog*, sous certaines conditions, perd les propriétés intéressantes de la Programmation en Logique telles que la déclarativité et la complétude, en raison :

- de l'utilisation d'une stratégie de recherche en profondeur d'abord,
- de l'influence de l'ordre d'apparition des clauses et l'agencement des littéraux au sein des clauses.
- du traitement de la négation logique : la négation basée sur l'hypothèse du monde clos doit être utilisée en *Prolog* car les clauses strictement négatives ne sont pas des clauses de Horn.
- de l'utilisation de prédicats "*extra-logiques*" comme le `cut`.

3 Programmation vs Démonstration

L'une des raisons de l'inefficacité de la Démonstration Automatique comparativement à la Programmation Logique, provient de l'explosion combinatoire due au fait que, dans le cadre du *principe de résolution*, chaque littéral d'une clause, combiné avec d'autres clauses, est susceptible de donner naissance à de nouvelles résolvantes. Par opposition, un système de Programmation en Logique ne produit pas de nouvelles résolvantes qui viendront s'ajouter aux axiomes initiaux du problème. En outre, la forme particulière des clauses de Horn permet de limiter le nombre de littéraux à partir desquels seront obtenus des résultats, en se limitant au seul littéral positif de la clause.

En prenant le contre-pied des affirmations précédentes, on peut dire que l'efficacité de *Prolog* comparativement aux systèmes de Démonstration Automatique, tient aux facteurs suivants :

- *les clauses de Horn instaurent un comportement procédural des clauses*, ce qui a pour effet de réduire considérablement l'espace de recherche.

  > « *In a Prolog program, clauses are used to express an algorithm chosen to solve some given problem in a very focused fashion – an algorithm intended to compute a solution with very little searching. In contrast, the clauses given to a typical automated reasoning program are used to initiate a less focused search for a solution – less focused in that no precise algorithm is employed to directly compute the solution.* » ([132], p. 487)

- la *monotonie* de la *résolution* SLD, qui constitue le coeur de *Prolog*, est déterminante. La *résolution* SLD peut être considérée comme de l'*hyper-résolution négative* ([110], [132], p. 489) (ou comme de la *résolution unitaire*) sans ajout de nouvelles clauses à l'ensemble de clauses initiales du problème. On a donc nul besoin de faire appel à des techniques coûteuses en temps comme la *subsumption* ou la *démodulation*,
- le *test d'occurrence*, également pénalisant en temps et utilisé en Démonstration Automatique lors de l'unification de deux termes, n'est pas utilisé en *Prolog*. Bien que l'absence de *test d'occurrence* puisse conduire à des déductions invalides, en pratique ce genre de situation ne se produit que très rarement.

La relative efficacité des systèmes de Programmation en Logique, "*capables d'effectuer des inférences à un niveau au moins deux fois supérieur par rapport aux systèmes de Démonstration Automatique*" [115], conduit **Stickel** à proposer en 1984 d'utiliser *Prolog* en tant que système de Démonstration Automatique en modifiant l'interpréteur *Prolog* pour remédier aux "défauts logiques" de ce dernier. Le système PTTP [116] (*Prolog Technology Theorem Prover*) qui en résulte est basé sur l'*élimination de modèles* [71]. En 1987, **Loveland** élabore le système *near-Horn Prolog* qui présente l'avantage de ne pas utiliser toutes les contrapositives d'une clause, à la différence de PTTP. Le système *near-Horn Prolog* est une extension minimale de *Prolog* qui donne la préférence à l'efficacité des traitements, plutôt qu'à la complétude. En 1988, **Plaisted** [94], à l'instar de Stickel et Loveland, propose d'étendre la Programmation en Logique afin qu'elle ne soit plus seulement spécifique aux clauses de Horn et qu'elle intègre la sémantique logique de la négation. Plus récemment, le démonstrateur SATCHMO [76] a pu être identifié dans [70] comme étant un démonstrateur *bidirectionnel* employant le *chaînage avant* sur les clauses non Horn et le *chaînage arrière* sur les clauses de Horn, comme le ferait *Prolog*.

La clé de voûte de ces systèmes repose sur la séparation entre les clauses de Horn et les clauses non Horn. Les clauses de Horn bénéficient de l'efficacité de l'interpréteur *Prolog* et compensent l'inefficacité du traitement des clauses non Horn.

Une autre différence essentielle concernant la Programmation en Logique et la Démonstration Automatique, évoquée précédemment, réside dans la différence des questions traitées. Ainsi, en Démonstration Automatique, le résultat escompté au sortir de la résolution est une réponse de type Oui/Non, témoignant de la consistance de l'ensemble des axiomes du problème, ce que **Chang** et **Lee** [23] identifient sous le nom de question de *classe A*. En revanche, le résultat de l'exécution d'un programme *Prolog*, qui est une question de *classe B*, représente l'ensemble des instances qui sont solutions du problème. En d'autres termes, *la résolution d'un problème dans le cadre de la Démonstration Automatique permet de déterminer l'existence d'une solution sans nécessairement devoir l'exhiber*. Considérons par exemple le problème de coloriage de la figure 3, qui consiste à colorier chaque rectangle de la figure avec l'une des trois couleurs rouge, vert, bleu, sans que deux rectangles adjacents soient de la même couleur.

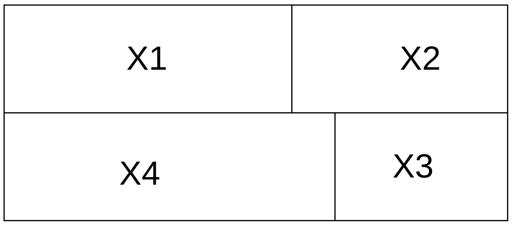

FIG. 3 – *Problème de coloriage*

La formulation de ce problème en *Prolog* est donnée par :

```
voisin( rouge, vert  ).
voisin( rouge, bleu  ).
voisin( vert,  rouge ).
voisin( vert,  bleu  ).
voisin( bleu,  rouge ).
voisin( bleu,  vert  ).
coloriage( X1, X2, X3, X4 ) :- voisin(X1,X2), voisin(X1,X4),
                               voisin(X2,X3), voisin(X2,X4),
                               voisin(X3,X4).
```

Le prédicat *voisin* définit les couples de couleurs autorisés. Ainsi $voisin(X,Y)$ signifie que deux zones adjacentes peuvent être coloriées, l'une dans la couleur X et l'autre en Y, X et Y étant des couleurs différentes.

A la question « *Existe-t-il un coloriage solution du problème* »,

```
? coloriage(X1,X2,X3,X4).
```

la réponse fournie par *Prolog* est : « *Oui, il en existe plusieurs, en voici la liste* »,

```
coloriage(bleu,vert,bleu,rouge).
coloriage(bleu,rouge,bleu,vert).
coloriage(vert,bleu,vert,rouge).
coloriage(vert,rouge,vert,bleu).
coloriage(rouge,bleu,rouge,vert).
coloriage(rouge,vert,rouge,bleu).
```

En comparaison, le même problème en Démonstration Automatique s'exprime de la manière suivante :

$$
\begin{array}{l}
voisin(rouge{,}vert).\\
voisin(rouge{,}bleu).\\
voisin(vert{,}rouge).\\
voisin(vert{,}bleu).\\
voisin(bleu{,}rouge).\\
voisin(bleu{,}vert).\\
coloriage(x_1{,}x_2{,}x_3{,}x_4) \vee \neg voisin(x_1{,}x_2) \vee \neg voisin(x_1{,}x_4) \vee\\
\quad \neg voisin(x_2{,}x_3) \vee \neg voisin(x_2{,}x_4) \vee \neg voisin(x_3{,}x_4)
\end{array}
$$

mais si l'on pose la même question que précédemment,

$$\neg coloriage(x_1{,}x_2{,}x_3{,}x_4)$$

la réponse généralement obtenue est : « *Oui, il existe un coloriage.* », témoignant de l'inconsistance de l'ensemble d'axiomes.

Si la Démonstration Automatique ne permet pas d'exhiber les solutions d'un tel problème, en revanche la Programmation en Logique en est capable. Nous verrons que le logiciel que nous avons conçu dans le cadre de cette thèse permet d'obtenir, à l'instar de *Prolog*, l'ensemble des solutions de ce problème.

En résumé, la Démonstration Automatique s'intéresse uniquement à la recherche de consistance d'un ensemble d'axiomes alors que la Programmation en Logique a pour vocation de calculer les solutions à une question précise. La distinction entre Programmation en Logique et Démonstration Automatique est susceptible d'être affinée en ces termes : si l'on cherche à démontrer la consistance d'un ensemble d'axiomes, on aura tout intérêt à examiner une grande partie de l'espace de recherche. Par contre, si l'on cherche à exhiber les interprétations en réponse à un but donné, il est préférable de se focaliser dans la direction menant à la réponse désirée en restreignant l'espace de recherche.

4 Programmation par Contraintes

La **Programmation par Contraintes** (PPC) est issue de la rencontre de deux domaines de recherche que sont l'**Intelligence Artificielle** en Informatique et la **Recherche Opérationnelle** en Mathématiques.

Elle est aujourd'hui le siège d'importantes recherches en raison de l'engouement qu'elle suscite dans les milieux industriels et financiers car elle permet d'attaquer des problèmes combinatoires complexes comme, entre autres, l'optimisation de la gestion des ressources [40] ou la planification de la production [6].

L'attrait que connaît la PPC est dû à deux facteurs principaux. Le premier de ces facteurs concerne la simplicité de modélisation des problèmes. Le paradigme des **Problèmes de Satisfaction de Contraintes**[6] (CSP), qui constitue le formalisme de représentation des problèmes de la PPC, ne demande aucune compétence dans le domaine de la programmation. Un CSP est décrit par un ensemble de variables évoluant chacune sur un domaine de valeurs (ou domaine de discours) et devant satisfaire un ensemble de relations (ou *contraintes*). Tout comme en Programmation en Logique, le formalisme CSP conduit l'utilisateur à se préoccuper uniquement de la composante *logique* sans se soucier du *contrôle*.

Le second facteur tient à l'efficacité des techniques de résolution mises en oeuvre pour résoudre les CSP. Les techniques de résolution CSP ont par exemple été appliquées en calcul propositionnel avec une très nette efficacité. Un certain nombre de logiciels performants ont ainsi vu le jour. Citons par exemple clp(FD) [26] et *Score* [21].

Le stage de DEA que j'ai effectué en 1993-1994, au sein du laboratoire LIRSIA[7], m'a permis de me familiariser avec les techniques de PPC et le formalisme CSP. L'objectif de ce stage consistait à implanter des techniques de symétries dans le cadre du logiciel *Score*(FD/B). Une symétrie [63] est une permutation

6. ou Constraint Satisfaction Problems
7. Laboratoire d'Informatique et de Recherche sur les Systèmes Intelligents et leurs Applications

de littéraux laissant invariant un ensemble de clauses. Les symétries offrent un procédé d'élagage de l'arbre de recherche par élimination de cas redondants (ou symétriques) infructueux lors de la résolution.

Les études menées par **Benhamou** et **Saïs** [7, 9, 8], puis par **Crawford** et al. [30], montrent que l'utilisation des propriétés de symétries inhérentes à certains types de problèmes permet d'améliorer de manière significative la résolution de ces problèmes, en l'occurrence les problèmes structurés insatisfiables, tels le problème de **Ramsey**, ou celui des pigeons.

En particulier, le problème de Ramsey(3,3,3) qui consiste à déterminer un coloriage en trois couleurs d'un graphe complet de n sommets, possède des solutions pour $3 \leq n \leq 16$. Pour $n \geq 17$, le problème n'admet aucune solution, mais les méthodes standard sont radicalement inefficaces pour en établir la preuve, car elles consistent, en quelque sorte, à tester toutes les configurations possibles, et n'en trouver aucune qui soit acceptable, pour pouvoir déterminer qu'il n'existe aucune solution au problème. On effectue donc un parcours total de l'arbre de recherche pour démontrer l'insatisfiabilité du problème.

Les méthodes qui obtiennent les meilleurs résultats sont celles qui sont aptes à réduire de manière drastique l'arbre de recherche, et les propriétés de symétries agissent dans ce sens. Ainsi en intégrant les symétries dans le cadre de *Score*(FD/B) nous avons été capables de résoudre le problème de Ramsey pour $n = 17$ en une seconde [101, 22], alors que sans tenir compte des symétries, la résolution de ce problème est quasiment inabordable.

La méthode *Score* a donné lieu à une habilitation à diriger des recherches [20] ainsi qu'à deux thèses soutenues en 1998. La thèse de Vincent **Juliard**[58, 21] traite de la méthode pour le domaine fini booléen. Isabelle **Blot-Thibaut** [13, 14], quant à elle, s'est intéressée à l'implantation de *Score* en domaine fini entier.

L'ensemble de ces recherches démontre que la méthode *Score* est efficace en domaine fini booléen. En revanche, en domaine fini entier, *Score* perd de son efficacité et est moins performant que d'autres approches.

5 Objectif

L'ensemble de nos travaux se situent dans la lignée des travaux de Stickel [115] et Plaisted [94], c'est-à-dire augmenter l'efficacité de la Démonstration Automatique. La poursuite de cet objectif nous a permis de procurer un cadre qui permette de faire de la Démonstration Automatique (*test de consistance* et *calcul de modèle*) et d'envisager des extensions vers la Programmation en Logique (*calcul de réponse correcte*).

L'intérêt des travaux conduits au cours de cette thèse a été de rapprocher Démonstration Automatique et Programmation en Logique via le *catalyseur* que constitue le paradigme des Problèmes de Satisfaction de Contraintes.

```
Problem Coloriage;

Clauses

  voisin( rouge, vert  ).
  voisin( rouge, bleu  ).
  voisin( vert,  rouge ).
  voisin( vert,  bleu  ).
  voisin( bleu,  rouge ).
  voisin( bleu,  vert  ).
  coloriage( X1, X2, X3, X4 ) | -voisin(X1,X2) | -voisin(X1,X4),
                              | -voisin(X2,X3) | -voisin(X2,X4) | -voisin(X3,X4).
Find coloriage(X1,X2,X3,X4).
```

FIG. 4 – *Problème de coloriage en Sacre : spécification logique*

Ce rapprochement n'est pas fortuit. D'une part, la dualité de représentation des problèmes de la logique des prédicats du premier ordre exprimés sous forme normale conjonctive (cf. fig. 4) et sous forme CSP (cf. fig 5) a éveillé notre curiosité. D'autre part, les problèmes rencontrés en Démonstration Automatique sont des problèmes combinatoires complexes et les techniques de résolution relatives aux CSP permettent de s'attaquer à des problèmes difficiles. L'idée maîtresse qui fonde la Programmation en Logique repose sur la notion procédurale des clauses qui fournit un mécanisme d'élagage de l'arbre de recherche qui n'est *a priori* pas utilisé en Démonstration Automatique. Nous pensons qu'il serait utile d'introduire un tel mécanisme de contrôle qui pourrait être explicitement formulé par l'utilisateur ou géré automatiquement par le système de résolution permettant ainsi d'orienter la recherche suivant une approche Programmation en Logique.

Sur la base de ces réflexions, nous nous sommes attelés à élaborer un nouveau cadre de travail qui permette de résoudre plus efficacement les problèmes de la logique des prédicats du premier ordre, spécifiés sous forme normale conjonctive.

Ce cadre de travail, baptisé *Sacre*, s'établit sur la base d'une modélisation des problèmes de la logique des prédicats sous forme de **Problèmes de Satisfaction de Contraintes** et nous permet de rechercher simultanément une réfutation et un modèle d'un problème.

Cette intrusion du formalisme des CSP à l'intérieur d'un cadre logique, ne revêt pas seulement un caractère technique, mais sous-tend l'utilisation de procédés qui ont montré leur capacité à résoudre des problèmes combinatoires complexes en calcul propositionnel, mais également dans d'autres domaines. À l'origine, il devait constituer une transposition de *Score*(FD/B) [21, 58] au calcul des prédicats. Notre contribution a alors consisté à élever ces techniques à l'ordre supérieur, c'est-à-dire, en calcul des prédicats. Si cet exercice semble simple de prime abord, il constitue néanmoins une pierre d'achoppement, révélant de nombreux obstacles dans la mesure ou l'on désire rester fidèle au formalisme des CSP. En conséquence, il est nécessaire de modifier le cadre CSP originel afin d'offrir la possibilité d'un traitement adéquat.

Le logiciel *Sacre*, qui est une implantation des concepts issus de nos travaux, accepte en entrée une spécification qui peut être formulée sous forme CNF (cf. fig. 4) ou sous forme de CSP (cf. fig 5).

```
Problem Coloriage;

Domains
  voisin+ = { ( rouge, vert ),  ( rouge, bleu ),
              ( vert,  rouge ), ( vert,  bleu ),
              ( bleu,  rouge ), ( bleu,  vert ) }

Constraints
  #(1,6, coloriage( X1, X2, X3, X4 ), -voisin(X1,X2), -voisin(X1,X4),
                                      -voisin(X2,X3), -voisin(X2,X4),
                                      -voisin(X3,X4) ).

Find coloriage(X1,X2,X3,X4).
```

FIG. 5 – *Problème de Coloriage en* Sacre *: spécification CSP*

Le résultat de la résolution de ce problème par *Sacre* permettra d'exhiber l'ensemble des coloriages possibles (voir fig 6) eprimés sous forme d'un domaine associé à une variable, en l'occurrence *coloriage*+.

6 Plan

Le plan que nous suivrons s'organise en deux parties. Dans la première partie de cette thèse nous abordons les notions essentielles qui constituent le bagage nécessaire à la compréhension des différents aspects de notre approche. Nous commençons par un rappel concernant *Calcul Propositionnel* et *Calcul des Prédicats*. Nous abordons également de manière détaillée, les procédés utilisés en Démonstration Automatique pour résoudre les problèmes de la logique des prédicats du premier ordre. Enfin, un chapitre sur les *Problèmes de Satisfaction de Contraintes* achève notre exposé introductif.

```
coloriage+ = { (bleu,vert,bleu,rouge),
               (bleu,rouge,bleu,vert),
               (vert,blue,vert,rouge),
               (vert,rouge,vert,bleu),
               (rouge,bleu,rouge,vert),
               (rouge,vert,rouge,bleu) }
```

FIG. 6 – *Solutions au problème de coloriage*

Dans la deuxième partie, nous présentons les aspects formels de la méthode de résolution que nous avons développé. Dans un premier temps, nous nous intéressons à l'opérateur de cardinalité car il est à la base de la modélisation sous forme de CSP des problèmes que nous sommes amenés à traiter. Nous montrons, en marge des résultats théoriques obtenus dans cette thèse, que son pouvoir expressif et sa forme particulière permettent de concevoir un système de déduction capable de résoudre avec efficacité certaines classes de problèmes en calcul propositionnel.

Puis, nous décrivons un procédé de résolution des problèmes de la logique des prédicats du premier ordre exprimés sous forme normale conjonctive par une approche qui se fonde sur une modélisation de ces problèmes en tant que Problèmes de Satisfaction de Contraintes. Ce procédé de résolution est complet et permet d'effectuer simultanément du test de consistance et du calcul de modèles.

Nous passons ensuite à la description de l'implantation du logiciel *Sacre* qui représente la mise en oeuvre de notre approche et comparons celui-ci à deux autres approches basées sur des procédés de résolution différents (Otter [80], Setheo [86]). Enfin nous clôturons notre exposé en présentant un ensemble de résultats comparatifs afin de prouver la validité de notre démarche.

Première partie

Prérequis

Chapitre 1

Logique et Calcul Propositionnel

« *La logique est la jeunesse des mathématiques et les mathématiques sont l'âge viril de la logique.* »

B. Russell, Introduction à la philosophie mathématique.

L'ensemble des résultats présentés dans les deux chapitres qui suivent sont développés de manière détaillée dans un support de cours [102].

1.1 Syntaxe

1.1.1 Définition informelle

La logique des propositions est construite sur la base d'un système formel dont nous donnons ici une définition. La *syntaxe* de ce système permet d'écrire des phrases dans une langue particulière, celle de la logique des propositions ou calcul propositionnel. La syntaxe permet de formaliser la notion de démonstration, qui en logique, est une suite de formules qui sont obtenues par l'intermédiaire d'un procédé mécanique [8] déterministe. Il s'agit en fait de **construire une théorie de la déduction.**

Une déduction met en jeu une prémisse, qui si elle est vraie, engendre une conclusion. Nous pouvons alors affirmer que nous tentons de formaliser des raisonnements du type : si *prémisse* alors *conclusion*, dont le schéma structurel prend la forme *si P alors C*.

Nous risquons alors la confusion entre les phrases du langage que nous construisons (qui contiennent des structures *si alors*) et notre propre langue (le français) qui utilisera probablement les mêmes structures pour décrire cette nouvelle langue.

Il semble alors évident que la construction de la logique en tant que langue pose un problème, à savoir, déterminer à quel niveau de langage on se trouve. Considérons, par exemple, la phrase suivante :

si la formule H est vraie et si l'on sait que si H alors F, alors on en déduit F.

On distingue dans cette phrase deux structures *si alors*. La première correspond à ce que **Tarski** appelle la *métalangue* (ou langue de l'observateur – dans le cas présent le français) utilisée pour construire les formules. Le deuxième *si alors* appartient à la *langue objet* (i.e. celle qui est prise comme objet d'une théorie) donc au calcul propositionnel. Nous utiliserons donc des représentations différentes suivant le niveau de langage. En particulier, *si P alors C* s'écrira $P \rightarrow C$ en calcul propositionnel. La phrase précédente s'énonce donc :

si la formule H est vraie et si l'on sait que si $H \rightarrow F$, alors on en déduit F.

Nous donnons à présent la syntaxe du calcul propositionnel sous forme d'un système formel.

8. le mot *mécanique* devant être compris ici au sens de *récursif*.

1.1.2 Définition formelle

Définition 1.1.1 - Alphabet - *L'alphabet du calcul propositionnel est défini par un ensemble de symboles répartis en :*

- symboles de vérité *:* $\top$ *et* $\bot$*,*
- variables propositionnelles *: un ensemble infini dénombrable* $V = \{p,q,r,\ldots\}$ *de variables propositionnelles ou atomes qui sont des énoncés dont on ne connaît pas la signification,*
- signes de ponctuation *: (et)*
- connecteurs logiques *: qui sont des opérateurs reliant les variables propositionnelles,*

Connecteur mono-adique	$\neg$	*négation*
Connecteurs di-adiques	$\vee$	*conjonction*
	$\wedge$	*disjonction*
	$\rightarrow$	*implication*
	$\equiv$	*équivalence*

Définition 1.1.2 - Formules bien formées - *L'ensemble des formules bien formées (fbf) du calcul propositionnel est défini inductivement par :*

- $\top$ *et* $\bot$ *sont des formules bien formées,*
- *les atomes sont des formules bien formées,*
- *si* A *est une formule bien formée , alors* (A), $(\neg A)$ *sont des formules bien formées,*
- *si* A *et* B *sont des formules bien formées, alors* $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \equiv B)$ *sont des formules bien formées,*
- *toute formule bien formée est obtenue par application des règles précédentes un nombre fini de fois.*

Ainsi, $((p \rightarrow q) \vee (p \equiv r))$ et $(((\neg p) \wedge \top) \vee ((q \rightarrow s) \vee r))$ sont des exemples de formules bien formées.

Définition 1.1.3 - Littéral - *On fait parfois la différence entre les atomes précédés du symbole de négation* $\neg$ *et ceux qui ne le sont pas : on qualifie les premiers de littéraux négatifs et les seconds de littéraux positifs.*

1.2 Axiomatique

L'axiomatique, c'est-à-dire les axiomes et les règles d'inférence, permet de distinguer les assemblages de mots qui font partie du langage, appelés *théorèmes*, de ceux qui n'en font pas partie appelés *non théorèmes*. En particulier, les axiomes sont des théorèmes.

Définition 1.2.1 - Axiomes - *Les axiomes du calcul propositionnel sont au nombre de trois :*

(A1)	$\vdash A \rightarrow (B \rightarrow A)$	*(conséquence de l'hypothèse)*
(A2)	$\vdash (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$	*(auto distributivité de* $\rightarrow$*)*
(A3)	$\vdash (\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$	*(contraposition partielle)*

L'axiome **(A3)** est parfois remplacé par $(\neg A \rightarrow B) \rightarrow ((\neg A \rightarrow \neg B) \rightarrow A)$.

Définition 1.2.2 - Règle d'inférence - *Le calcul propositionnel utilise une seule règle d'inférence appelée règle du* Modus Ponens *également qualifiée de règle de* détachement *:*

$$\frac{\vdash A \qquad \vdash (A \rightarrow B)}{\vdash B}\ (MP)$$

qui peut se lire : si A *est un théorème et si* $A \rightarrow B$ *est aussi un théorème alors* B *est un théorème*

En fait, on utilise également une autre règle, dite règle de *substitution* mais celle-ci n'est généralement pas énoncée directement, puisque profondément enfouie dans la structure des axiomes. Les axiomes sont en fait des *schémas d'axiomes*, c'est à dire que les lettres A,B,C représentent des formules quelconques. Soient p,q deux variables propositionnelles. En remplaçant A par p et B par q dans **(A1)**, on obtient $p \rightarrow (q \rightarrow p)$. De même, en remplaçant A par q et B par p, on a $q \rightarrow (p \rightarrow q)$, etc.

Définition 1.2.3 - Démonstration - *Une démonstration d'un théorème A est une suite finie $(L_1,L_2,\ldots,L_n)$ où chaque L_i est soit un axiome, soit obtenu par l'application de la règle d'inférence du Modus Ponens sur les théorèmes L_j et L_k précédemment produits $(j,k < i)$. On note alors $\vdash A$ qui peut se lire A est un théorème.*

1.2.1 Notion de déduction

Nous le notions en introduction de ce chapitre : il s'agit pour nous d'élaborer une théorie de la déduction. Le théorème suivant va nous permettre d'atteindre notre but en formalisant la notion de déduction.

Définition 1.2.4 - Déduction - *Une formule F se déduit d'un ensemble de formules $H = (H_1,H_2,\ldots,H_n)$, s'il existe une suite finie $(L_1,L_2,\ldots,L_i,\ldots,L_m,F)$ où chaque L_i est soit un axiome, soit un des H_i, soit obtenu par l'application de la règle du* Modus Ponens *sur deux éléments L_j,L_k de la suite déjà obtenue $(j,k < i)$. Dans ce cas, on note $H \vdash F$.*

Ainsi, la formule $p \rightarrow r$ se déduit de $H = \{(p \rightarrow q),(q \rightarrow r)\}$:

1	$\vdash p \rightarrow q$	
2	$\vdash q \rightarrow r$	
3	$\vdash ((p \rightarrow q) \rightarrow r) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$	Axiome **(A2)** avec $A = p,B = q,C = r$
4	$\vdash (q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$	Axiome **(A1)** avec $A = q \rightarrow r,B = p$
5	$\vdash p \rightarrow (q \rightarrow r)$	$MP(2,4)$
6	$\vdash (p \rightarrow q) \rightarrow (p \rightarrow r)$	$MP(5,3)$
7	$\vdash p \rightarrow r$	$MP(1,6)$
	$\{p \rightarrow q,q \rightarrow r\} \vdash p \rightarrow r$	

Théorème 1.2.1 - Théorème de déduction formelle - *Soit H un ensemble de formules et $G \in H$. $H \vdash F \iff H \setminus \{G\} \vdash G \rightarrow F$. En particulier,*

$$H \vdash F \iff \vdash H \rightarrow F$$

Définition 1.2.5 - Théorie - *On appelle théorie le système formel composé du système formel du calcul propositionnel dont les axiomes sont augmentés par un ensemble de formules H. Les formules de H constituent le fondement de la théorie et sont donc tenues comme vraies.*

Nous allons prendre un exemple assez simpliste pour illustrer notre propos. Considérons que nous ayons pu noter que les faits suivants se répétaient assez fréquemment :

- s'il fait beau alors je fais du vélo,
- si je fais du vélo alors il y a du vent,

Nous voudrions étudier ces faits et déterminer s'il existe une relation entre-eux. Nous allons pour cela construire une théorie T à partir des faits observés. Convenons de remplacer *il fait beau* par la variable propositionnelle p, *je fais du vélo* par q et *il y a du vent* par r. La représentation des faits en tant que formules logiques donne respectivement : $p \rightarrow q$ et $q \rightarrow r$. En utilisant les résultats précédents, nous pouvons conclure $p \rightarrow r$, soit *s'il fait beau alors il y a du vent.* Ce qui constitue, somme toute, une loi météorologique discutable à l'échelle de la réalité, mais valide du point de vue de la théorie.

1.3 Sémantique

Nous devons à présent donner un sens aux formules afin de pouvoir les interpréter. Pour celà nous allons commencer par donner un sens aux variables propositionnelles qui apparaissent dans les formules : nous associons donc à chaque variable, une *valeur de vérité* qui est soit la valeur t (*vrai*), soit la valeur f (*faux*). En particulier, le symbole $\top$ a pour valeur de vérité t, et $\bot$ a pour valeur f.

Définition 1.3.1 - Valuation - *On appelle valuation d'un ensemble de variables propositionnelles $W \subseteq V$, une application de W dans $\{t,f\}$ (voir figure 1.1.)*

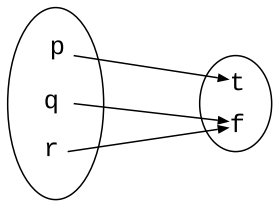

FIG. 1.1 – *Exemple de valuation $p = t, q = f, r = f$*

Définition 1.3.2 - Interprétation - *Une interprétation I d'une formule F est une valuation des variables de F.*

On associe ensuite à chaque connecteur logique, une table de vérité, qui détermine de manière univoque la valeur de vérité du connecteur en fonction de ses opérandes.

Définition 1.3.3 - Fonction d'interprétation - *On appelle fonction d'interprétation ρ, une fonction qui évalue la valeur de vérité d'une formule en fonction des connecteurs logiques (cf. table 1.1) et d'une interprétation de la formule.*

p	q	$\neg p$	$\neg q$	$p \to q$	$p \vee q$	$p \wedge q$	$p \equiv q$
f	f	t	t	t	f	f	t
f	t	t	f	t	t	f	f
t	f	f	t	f	t	f	f
t	t	f	f	t	t	t	t

TAB. 1.1 – *Interprétation des connecteurs usuels*

Soit I une interprétation d'une formule F. On note $\rho(F,I)$, le résultat de l'évaluation de F par I.

Par exemple, l'interprétation de la formule $F = ((p \vee q) \to (q \wedge r))$ pour l'interprétation $I = \{p = t, q = t, r = f\}$ est :

1. $F_1 = (p \vee q)$, $\rho(F_1,I) = t$,
2. $F_2 = (q \wedge r)$, $\rho(F_2,I) = f$,
3. $F = (F_1 \to F_2)$, $\rho(F,I) = f$.

Définition 1.3.4 - Modèle d'une formule - *Un modèle d'une formule F est une interprétation I de F telle que $\rho(F,I) = t$. Si I est un modèle de F, on note $I \models F$.*

Définition 1.3.5 - Formule satisfiable (ou consistante) - *Une formule F est dite satisfiable ou consistante, s'il existe une interprétation I de F telle que $I \models F$. En d'autres termes, une formule satisfiable possède au moins un modèle.*

Définition 1.3.6 - Tautologie (ou formule valide) - *Une tautologie ou formule valide est une formule F vraie dans toute interprétation, quelles que soient les valeurs de vérité des atomes qui la composent. On note alors $\models F$.*

En particulier, les axiomes sont des tautologies.

Définition 1.3.7 - Formule insatisfiable - *Une formule fausse dans toute interprétation est appelée antitautologie ou est dite insatisfiable.*

Si F est une tautologie, alors $\neg F$ est une formule insatisfiable et inversement.

Exemples :

$(p \wedge \neg p)$	est insatisfiable
$(p \vee \neg p)$	est une tautologie ou formule valide
$(p \rightarrow (\neg p))$	est consistante

Définition 1.3.8 - Ensemble insatisfiable de formules - *Un ensemble H de formules est insatisfiable, si et seulement si il n'existe aucune interprétation I de H telle que toutes les formules de H soient satisfaites par I.*

Définition 1.3.9 - Conséquence Valide - *Soit une formule F et un ensemble de formules H. On dit que F est conséquence valide de H si tout modèle de H est modèle de F. On note alors $H \models F$.*

Théorème 1.3.1 - Théorème sémantique de la déduction - *Soit F une formule et H un ensemble de formules.*

$$H \models F \iff \models H \rightarrow F$$

Théorème 1.3.2 - Théorème de déduction par réfutation - *Soit F une formule et H un ensemble de formules, alors*

$$H \models F \iff H \wedge \neg F \text{ est insatisfiable}$$

Ce théorème est à la base de nombreuses méthodes de résolution des problèmes de la logique.

Théorème 1.3.3 - Théorème de compacité - *Soit H un ensemble de formules. Si toute partie finie de H est satisfiable alors H est satisfiable.*

Corollaire 1.3.1 - *Un ensemble de formules H est insatisfiable si une partie finie de H est insatisfiable.*

1.4 Propriétés fondamentales

Le calcul propositionnel jouit d'un certain nombre de propriétés relatives aux systèmes formels. Nous reprenons ici les définitions données dans [1] :

1.4.1 Consistance

Définition 1.4.1 - Consistance - *Une logique est dite (syntaxiquement) consistante s'il n'existe aucune formule F telle que $\vdash F$ et $\vdash \neg F$.*[9]

Théorème 1.4.1 - Consistance du calcul propositionnel - *Le calcul propositionnel est consistant.*

En particulier la formule bien formée $\neg(p \rightarrow p)$ n'est pas un théorème car $p \rightarrow p$ est un théorème (appelé $(T1)$ dans ce chapitre).

1.4.2 Adéquation

Définition 1.4.2 - Adéquation - *Une logique est dite adéquate*[10] *si tout théorème $(\vdash F)$ est une formule valide $(\models F)$.*

Théorème 1.4.2 - Adéquation du calcul propositionnel - *Le calcul propositionnel est adéquat.*

9. La propriété de consistance s'énonce également en disant qu'il existe des formules bien formées qui ne sont pas des théorèmes.

10. *sound* en anglais.

1.4.3 Complétude

Définition 1.4.3 - Complétude faible - *Une logique est dite faiblement complète si toute formule valide* ($\models F$) *est un théorème* ($\vdash F$).

Définition 1.4.4 - Complétude forte - *Une logique est dite fortement complète ssi la relation de conséquence valide correspond à la relation de déduction : Soit* H *un ensemble de formules, si* $H \models F$ *alors* $H \vdash F$.

Théorème 1.4.3 - Complétude du calcul propositionnel - *Le calcul propositionnel est fortement complet.*

En d'autres termes, un calcul est dit fortement complet si l'adjonction à des schémas d'axiomes d'un schéma de formule qui n'est pas démontrable, le rend non consistant. Le théorème de complétude forte établit l'équivalence entre syntaxe et sémantique, ce qui signifie que $\vdash$ et $\models$ sont interchangeables.

Remarquons qu'il ne faut pas confondre la notion de complétude précédente avec la notion de complétude syntaxique et celle de complétude pour une procédure de preuve.

Définition 1.4.5 - Complétude syntaxique - *Une logique est dite syntaxiquement complète ssi, pour toute formule* F, *on a soit* $\vdash F$, *soit* $\vdash \neg F$.

Théorème 1.4.4 - Incomplétude syntaxique du calcul propositionnel - *Le calcul propositionnel est syntaxiquement incomplet.*

1.4.4 Décidabilité

Définition 1.4.6 - Décidabilité - *Une logique est décidable s'il existe une procédure mécanique qui permet d'établir en un temps fini si une formule est ou non un théorème.*

Théorème 1.4.5 - Décidabilité du calcul propositionnel - *Le calcul propositionnel est décidable.*

1.5 Formes normales

Les formes normales partent du principe que les connecteurs $\rightarrow$ et $\equiv$ peuvent s'exprimer de manière équivalente en fonction de $\wedge$ et $\vee$ qui sont des connecteurs avantageux puisque commutatifs et associatifs. En conséquence, dans une conjonction ou une disjonction de littéraux l'ordre d'apparition des littéraux et la disposition des parenthèses n'ont pas d'importance.

Définition 1.5.1 - Formules équivalentes - *Deux formules sont dites équivalentes si elles possèdent la même table de vérité.*

Définition 1.5.2 - Clause - *Une clause est une disjonction de littéraux de la forme* $l_1 \vee \ldots \vee l_m$.

Définition 1.5.3 - Clause unitaire - *Une clause unitaire est une clause composée d'un seul littéral.*

Définition 1.5.4 - Clause de Horn - *Une clause de Horn est une clause composée au maximum d'un littéral positif et d'au moins un littéral négatif.*

Définition 1.5.5 - Forme normale conjonctive - *Une formule* F *est sous forme normale conjonctive (FNC)*[11] *lorsqu'elle s'écrit* $F = F_1 \wedge \ldots \wedge F_n$, *où chaque* F_i, *est une clause.*

Définition 1.5.6 - Forme normale disjonctive - *Une formule* F *est sous forme normale disjonctive (FND)*[12] *lorsqu'elle s'écrit* $F = F_1 \vee \ldots \vee F_n$, *où chaque* F_i, $1 \leq i \leq n$ *est une conjonction de littéraux de la forme* $F_i = l_1 \wedge \ldots \wedge l_m$.

11. ou CNF pour Conjunctive Normal Form.
12. ou DNF pour Disjunctive Normal Form.

Théorème 1.5.1 - Théorème de normalisation - *Toute formule admet une forme normale conjonctive et disjonctive qui lui est logiquement équivalente.*

La transformation d'une formule sous forme normale (conjonctive ou disjonctive) se réalise facilement par application des règles d'associativité, distributivité et commutativité des connecteurs $\wedge$ et $\vee$.

Théorème 1.5.2 - FNC et validité - *La mise sous forme normale conjonctive est un procédé de décision pour la recherche de validité d'une formule.*

Théorème 1.5.3 - FND et insatisfiabilité - *La mise sous forme normale disjonctive est un procédé de décision pour la recherche d'insatisfiabilité d'une formule.*

Chapitre 2

Calcul des Prédicats

« *On a dit que Dieu pouvait tout créer, sauf ce qui serait contraire aux lois de la logique. En fait, nous ne saurions dire d'un monde "illogique" à quoi il ressemblerait.* »

L. WITTGENSTEIN, Tractatus logico-philosophicus 3.031

2.1 Introduction

Il est clair que le calcul propositionnel est insuffisant pour formaliser les raisonnements mathématiques. On est, par exemple, incapable d'exprimer en calcul propositionnel des expressions comme :

(Ph1) "*l'ensemble des X tels que X vérifie la propriété p.*"

(Ph2) "*le produit de X par son inverse est égal à l'identité.*"

C'est pourquoi, nous allons compléter et étendre le calcul propositionnel, en ce que l'on appelle un langage du premier ordre, en introduisant trois notions :

- la notion de *quantification*: elle permet de formaliser les expressions du type "*tous les X*" (quantification universelle, symbole $\forall$) ou du type "*il existe X*" (quantification existentielle, symbole $\exists$),
- la notion de *prédicat*: la phrase "*X vérifie la propriété p*" sera traduite par $p(X)$, où p est appelé symbole de prédicat,
- la notion de *fonction*: la notion de prédicat n'est pas suffisante pour exprimer la phrase *(Ph2)*. Nous adjoindrons donc à notre nouveau langage, des symboles de fonctions. La phrase *(Ph2)*, soit $X \times X^{-1} = e$, se traduit par la formule suivante $p(X,f(X),e)$, où p est un symbole de prédicat tel que $p(X,Y,Z)$ signifie le produit de X par Y est égal à Z ; e est le symbole de l'élément neutre pour la multiplication et f est un symbole de fonction représentant l'inverse.

Il nous faut attirer l'attention du lecteur sur le fait suivant : il existe plusieurs sortes de langages du premier ordre chacun de ces langages étant adapté à une famille déterminée de structures mathématiques en fonction de son vocabulaire. Cependant ces langages obéissent aux mêmes propriétés syntaxiques et sémantiques. Chaque langage est défini par trois ensembles de symboles : C l'ensemble des symboles de constantes, F l'ensemble des symboles de fonctions, P l'ensemble des symboles de prédicats. Par exemple, le langage des ensembles est défini par $C = \{\emptyset\}$, $F = \{\}$, $P = \{\in , =\}$. L'arithmétique de **Peano** est définie par les ensembles de symboles : $C = \{0,1\}$, $F = \{succ, + ,\times\}$, $P = \{< , =\}$.

2.2 Syntaxe

Nous donnons, dans cette section, la syntaxe d'un langage du premier ordre :

Définition 2.2.1 - alphabet - *L'alphabet d'un langage du premier ordre $L = \langle V,C,F,P\rangle$ est composé de plusieurs ensembles de symboles disjoints deux à deux et classés en :*

– Symboles de vérité *:* $\top$, $\bot$,

- constantes *: un ensemble fini (éventuellement vide)* $C = \{a,b,c,\ldots\}$ *de constantes,*
- variables *: un ensemble infini dénombrable* $V = \{X,Y,Z,\ldots\}$ *de variables,*
- symboles de fonctions *: un ensemble fini (éventuellement vide)* $F = \{f,g,h,\ldots\}$ *de symboles de fonction. Chaque symbole de fonction prend en paramètres un nombre d'arguments* n*, appelé* arité *du symbole de fonction, tel que* $n > 0$*,*
- symboles de prédicats *: un ensemble fini* $P = \{p,q,r,\ldots\}$ *de symboles de prédicats. Chaque symbole de prédicat prend en paramètres un nombre d'arguments (éventuellement nul)* n*, appelé* arité *du symbole de prédicat. Un prédicat d'arité nulle est appelé* variable propositionnelle *(à ne pas confondre avec l'ensemble des variables* V*),*
- connecteurs logiques *: qui sont des opérateurs*

Connecteur mono-adique	$\neg$	*négation*
Connecteurs di-adiques	$\vee$	*conjonction*
	$\wedge$	*disjonction*
	$\rightarrow$	*implication*
	$\equiv$	*équivalence*

- quantificateurs[13] *:*

$\exists$	*quantificateur existentiel*
$\forall$	*quantificateur universel*

Définition 2.2.2 - Terme - *L'ensemble* T_L *des termes d'un langage du premier ordre* $L = \langle V,C,F,P \rangle$ *est défini inductivement par :*

- *une constante* $c \in C$ *est un terme,*
- *une variable* $X \in V$ *est un terme,*
- *si* $f \in F$ *est un symbole de fonction d'arité* n *et si* $t_1,\ldots,t_n$ *sont des termes, alors* $f(t_1,\ldots,t_n)$ *est un terme,*
- *tout terme est obtenu par application des règles précédentes un nombre fini de fois.*

Définition 2.2.3 - Terme totalement instancié - *Un terme* $t \in T_L$ *est dit totalement instancié*[14] *s'il est composé uniquement de symboles de constantes et de fonctions. Autrement dit, s'il ne contient pas de variable.*

Définition 2.2.4 - Atome - *Un atome est défini inductivement par :*
- $\top$ *et* $\bot$ *sont des atomes,*
- *tout symbole de prédicat* $p \in P$ *d'arité* 0 *est un atome,*
- *si* $p \in P$ *est un symbole de prédicat d'arité* $n > 0$ *et si* $t_1,t_2,\ldots,t_n \in T_L$ *sont des termes, alors* $p(t_1,t_2,\ldots,t_n)$ *est un atome.*
- *tout atome est obtenu par application des règles précédentes un nombre fini de fois.*

Définition 2.2.5 - Atome totalement instancié - *Un atome est dit totalement instancié s'il ne contient pas de symbole de variable.*

Définition 2.2.6 - Formules bien formées - *L'ensemble* F_L *des formules bien formées d'un langage* $L = \langle V,C,F,P \rangle$ *est défini inductivement par :*

- *un atome est une formule bien formée,*
- *si* A *est une formule bien formée, alors* (A)*,* $(\neg A)$ *sont des formules bien formées,*
- *si* A *et* B *sont des formules bien formées, alors* $(A \wedge B)$*,* $(A \vee B)$*,* $(A \rightarrow B)$*,* $(A \equiv B)$ *sont des formules bien formées,*

13. appelés également quanteurs
14. ground en anglais.

- *si $A \in F_L$ est une formule bien formée et $X \in V$ une variable, alors*
  - $(\forall X A)$, *qui se lit : pour tout X on a A,*
  - $(\exists X A)$, *qui se lit : il existe X tel qu'on ait A,*

  sont des formules bien formées.
- *toute formule bien formée est obtenue par application des règles précédentes un nombre fini de fois.*

Définition 2.2.7 - Littéral - *On fait parfois la différence entre un atome précédé ou non du signe de négation $\neg$. Dans ce cas, on qualifie*
- $p(t_1,\ldots,t_n)$ *de littéral positif,*
- $\neg p(t_1,\ldots,t_n)$ *de littéral négatif„*

2.3 Variables libres, variables liées, portée et renommage

L'introduction des variables quantifiées soulève deux épineux problèmes. Le premier concerne la portée des quantificateurs. Celle-ci influe sur le sens donné aux formules. Bien que dans le cadre de la syntaxe, on fasse fi du sens, ce ne peut être le cas ici, car on trahirait alors le sens des formules. Le second problème découle directement du premier et touche aux occurrences libres et liées des variables dans les formules.

2.3.1 Variables libres, variables liées

Définition 2.3.1 *Soit F une formule dans laquelle apparaît la variable X. On notera alors $F[X]$ pour souligner le fait que X apparaît dans F.*

Définition 2.3.2 - Variables liées - *L'ensemble des variables liées d'une formule F, noté $V_b(F)$, est défini par :*
- *si F est un atome, $V_b(F) = \emptyset$,*
- *si F est de la forme $\neg G$, alors $V_b(F) = V_b(G)$,*
- *si F est de la forme $G \wedge H$, $G \vee H$, $G \rightarrow H$ ou $G \equiv H$, alors $V_b(F) = V_b(G) \cup V_b(H)$,*
- *si F est de la forme $(\forall X G)$ ou $(\exists X G)$, alors $V_b(F) = \{X\} \cup V_b(G)$*

Définition 2.3.3 - Variables libres - *L'ensemble des variables libres d'une formule F, noté $V_f(F)$, est défini par :*
- *si F est un atome, $V_f(F)$ est égal à l'ensemble des variables de F,*
- *si F est de la forme $\neg G$, alors $V_f(F) = V_f(G)$,*
- *si F est de la forme $G \wedge H$, $G \vee H$, $G \rightarrow H$ ou $G \equiv H$, alors $V_f(F) = V_f(G) \cup V_f(H)$,*
- *si F est de la forme $(\forall X G)$ ou $(\exists X G)$, alors $V_f(F) = V_f(G) - \{X\}$.*

Pour la formule $F = \forall X \exists Y p(X,Y,Z)$, on a $V_b(F) = \{X,Y\}$ et $V_f = \{Z\}$.

Définition 2.3.4 - Formule close - *Une formule bien formée F est dite close ou fermée si elle ne contient pas de variables libres ($V_f(F) = \emptyset$), sinon elle est dite ouverte. Pour toute formule ouverte F composée d'un ensemble de variables libres $X_1,\ldots,X_k$, on appelle*
- clôture universelle *la formule* $\forall X_1 \ldots \forall X_k\, F[X_1,\ldots,X_k]$,
- clôture existentielle *la formule* $\exists X_1 \ldots \forall X_k\, F[X_1,\ldots,X_k]$.

2.3.2 Portée

Définition 2.3.5 - Portée d'un quantificateur - *On appelle portée d'un quantificateur (ou scope), la sous-formule qui suit immédiatement le quantificateur.*

Ainsi, dans la formule $(\exists Z(\forall X p(X)) \rightarrow (\exists Y q(Z,Y)))$, la portée de $\forall X$ est $p(X)$, et la portée de $\exists Y$ est $q(Z,Y)$. La portée de Z est celle de la formule dans sa totalité.

L'ordre d'apparition des quantificateurs dans une formule est important. Il détermine le sens de la formule. Si nous donnons au prédicat $p(X,Y)$ la signification « X aime Y », alors, nous obtenons des significations différentes (voir table 2.1) pour la formule $\mathcal{Q}_1 X \mathcal{Q}_2 Y\, p(X,Y)$ (où $\mathcal{Q}_1$ et $\mathcal{Q}_2$ sont des quantificateurs).

$\forall X\,\forall Y\,p(X,Y)$	tout le monde aime tout le monde
$\exists X\,\forall Y\,p(X,Y)$	il existe des personnes qui aiment tout le monde
$\exists Y\,\forall X\,p(X,Y)$	il existe des personnes aimées de tous
$\forall X\,\exists Y\,p(X,Y)$	toute personne aime quelqu'un
$\forall Y\,\exists X\,p(X,Y)$	toute personne est aimée par quelqu'un
$\exists X\,\exists Y\,p(X,Y)$	quelqu'un aime quelqu'un

TAB. 2.1 – *Influence des quantificateurs sur le sens des formules*

2.3.3 Renommage

Considérons les deux formules suivantes :

$$\begin{aligned} F_1 = &\quad (\forall X p(X,Y)) \rightarrow (\forall X \exists Y\, q(X,Y)) \\ F_2 = &\quad (\forall X p(X,T)) \rightarrow (\forall Z \exists Y\, q(Z,Y)) \end{aligned}$$

Ces deux formules sont équivalentes bien qu'elles ne possèdent pas les mêmes noms de variables. En effet, la variable T située à gauche du connecteur $\rightarrow$ dans la formule F_2 n'est pas quantifiée. C'est donc une variable libre tout comme le premier Y dans F_1.

En outre, la portée du quantificateur $\forall X$ situé à droite du connecteur $\rightarrow$ dans la formule F_1 se limite à l'atome $p(X,Y)$. De même, la portée du quantificateur $\forall X$ situé à gauche du connecteur $\rightarrow$ toujours dans F_1 se limite à $q(X,Y)$. Le fait d'avoir deux fois le même nom de variable n'a aucune incidence sur le sens général de la formule, mais on préférera, pour des raisons de lisibilité, *renommer* une de ces variables (en Z dans F_2) pour ne pas les confondre.

2.4 Axiomatique

Nous entamons, à présent, la description axiomatique du calcul des prédicats.

Théorème 2.4.1 - Théorème de complétude de Gödel *($1^{ère}$ forme)* **-** *Pour chaque langage du premier ordre il existe une axiomatique, appelée calcul des prédicats, dont les théorèmes sont précisément les formules valides du langage.*

Définition 2.4.1 - Axiomes - *Les schémas d'axiomes du calcul des prédicats du premier ordre sont les suivants :*

(A1)	$\vdash A \rightarrow (B \rightarrow A)$	*(conséquence de l'hypothèse)*
(A2)	$\vdash (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$	*(auto distributivité de $\rightarrow$)*
(A3)	$\vdash (\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$	*(contraposition partielle)*
(A4)	$\vdash \forall x A(x) \rightarrow A(t)$	*(particularisation)*
(A5)	$\vdash (\forall x (A \rightarrow B)) \rightarrow (A \rightarrow (\forall x B))$	*(distributivité de $\forall$)*

On constate que les axiomes **(A1)**, **(A2)**, **(A3)** du calcul propositionnel sont toujours valables en calcul des prédicats.

Définition 2.4.2 - Règles d'inférence - *Le calcul des prédicats utilise deux règles d'inférence.*

- *le* Modus Ponens

$$\frac{\vdash A \qquad \vdash (A \rightarrow B)}{\vdash B}\ (MP)$$

- *la règle de généralisation :*

$$\frac{\vdash \forall x A}{\vdash A}\ (Gen)$$

Le théorème de la déduction formelle énoncé dans le chapitre sur le calcul propositionnel reste valable (voir théorème 1.2.1).

2.5 Substitution

Le remplacement d'une variable par une expression est une opération largement utilisée en mathématiques. Nous montrons dans cette section comment se définit la substitution d'une variable par un terme pour les formules d'un langage du premier ordre.

Définition 2.5.1 - Substitution - *Une substitution est une fonction d'un sous-ensemble de variables vers un sous-ensemble de termes $\sigma : W \rightarrow T_L$ où $W \subset V$ est appelé domaine de σ (noté $dom(\sigma)$). On notera plus simplement une substitution de la manière suivante : $\sigma = \{X_1 \hookleftarrow t_1, \ldots, X_k \hookleftarrow t_k\}$ où les t_i sont des termes de T_L et les X_i des variables de V.*

On peut également représenter une substitution sous la forme suivante :

$$\sigma = \{\frac{t_1}{X_1}, \ldots, \frac{t_k}{X_k}\}$$

Définition 2.5.2 - Substitution d'un terme - *L'application de la substitution σ à un terme t, notée $\sigma(t)$, est définie inductivement par :*

- *si $t = c$ alors $\sigma(c) = c$,*
- *si $t = X_i$ et $X_i \in dom(\sigma)$ alors $\sigma(X_i) = t_i$,*
- *si $t = X_i$ et $X_i \notin dom(\sigma)$ alors $\sigma(X_i) = X_i$,*
- *si $t = f(t_1, \ldots, t_n)$ alors $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.*

Le terme $\sigma(t)$ résultat de l'application de σ à t est appelé instance de t.

Par exemple, soit un terme $t = f(X,Y,c)$ et une substitution $\sigma = \{g(R)/X, b/Z\}$, alors $\sigma(t) = f(g(R),Y,c)$.

Définition 2.5.3 - Substitution d'une formule - *L'application d'une substitution σ à une formule F est définie inductivement par :*

- *si $F = p(t_1, \ldots, t_n)$ alors $\sigma(F) = p(\sigma(t_1), \ldots, \sigma(t_n))$,*
- *si $F = \neg G$ alors $\sigma(F) = \neg\sigma(G)$,*
- *si F est de la forme $G \otimes H$, où $\otimes$ est un connecteur di-adique, alors $\sigma(F) = \sigma(G) \otimes \sigma(H)$,*
- *si F est de la forme $(\forall X G)$, alors $\sigma(F) = (\forall X \sigma(G))$,*
- *si F est de la forme $(\exists X G)$, alors $\sigma(F) = (\exists X \sigma(G))$,*

On note également $F[X \hookleftarrow t]$ la formule F obtenue par substitution de la variable X par le terme t.

Définition 2.5.4 - Composition de substitutions - *Soient $\sigma = \{X_1 \hookleftarrow t_1, \ldots, X_n \hookleftarrow t_n\}$ et $\theta = \{Y_1 \hookleftarrow s_1, \ldots, Y_m \hookleftarrow s_m\}$ deux substitutions. La composition de θ et σ est la substitution $\sigma \circ \theta$ définie par :*

$$\sigma \circ \theta = \{\frac{\theta(t_1)}{X_1}, \ldots, \frac{\theta(t_n)}{X_n}, \frac{s_1}{Y_1}, \ldots, \frac{s_m}{Y_m}\}$$

dont on a supprimé tous les $Y_i \hookleftarrow \theta(t_i)$ pour lesquels $\theta(t_i) = Y_i$ et tous les $Y_i \hookleftarrow s_i$ tels que $Y_i \in \{X_1, \ldots, X_n\}$.

Exemple, soit $\sigma = \{\frac{f(Y)}{X}, \frac{Z}{Y}\}$ et $\theta = \{\frac{a}{X}, \frac{b}{Y}, \frac{Y}{Z}\}$, alors $\theta \circ \sigma = \{\frac{f(b)}{X}, \frac{Y}{Z}\}$

Proposition 2.5.1 - Associativité - *La composition de substitutions est associative. Soient un terme t et $\sigma_1, \sigma_2, \sigma_3$ trois substitutions. Alors, $((\sigma_1\sigma_2)\sigma_3)(t) = (\sigma_1(\sigma_2\sigma_3))(t)$.*

2.6 Sémantique

Définition 2.6.1 - Pré-interprétation - *Une pré-interprétation $I = \langle D,\rho_C,\rho_F\rangle$ d'une formule F est caractérisée par la donnée :*

- *d'un ensemble non vide D, appelé domaine de la pré-interprétation,*
- *d'une application ρ_C, qui à chaque symbole de constante $c \in C$ associe un élément de D,*
- *d'une application ρ_F qui à chaque symbole de fonction $f \in F$ d'arité n associe le graphe d'une fonction de $D^n \to D$ définissant f,*

Définition 2.6.2 - Interprétation des termes - *Étant donnée une pré-interprétation $I = \langle D,\rho_C,\rho_F\rangle$ d'une formule F, on note $\rho_T(t,I)$ la fonction d'interprétation d'un terme, définie de la manière suivante :*

- *si $t \in C$, $\rho_T(t,I) = \rho_C(t)$,*
- *si $t = f(t_1,\ldots,t_m) \in T_L$, $\rho_T(t,I) = \rho_F(f(\rho_T(t_1),\ldots,\rho_T(t_n)))$.*

Définition 2.6.3 - Interprétation - *Une Interprétation $\mathcal{I} = \langle I,\rho_P\rangle$ d'une formule F est caractérisée par la donnée :*

- *d'une pré-interprétation $I = \langle D,\rho_C,\rho_F\rangle$,*
- *d'une application ρ_P qui à chaque symbole de prédicat $p \in P$ d'arité n associe le graphe d'une fonction de $D^n \to \{t,f\}$ définissant p,*

Définition 2.6.4 - Interprétation des formules - *L'interprétation d'une formule F par une interprétation $\mathcal{I}$ est définie par la fonction d'interprétation $\rho(F,\mathcal{I})$, telle que :*

- *si $F = p(t_1,\ldots,t_n)$ est un atome, alors $\rho(F,\mathcal{I}) = \rho_P(\,p(\rho_T(t_1),\ldots,\rho_T(t_n))\,)$*
- *si F est de la forme $\neg G$, $G \wedge H$, $G \vee H$, $G \to H$, $G \equiv H$, $\rho(F,I)$ est définie comme en calcul propositionnel,*
- *si $F = \exists X G$, alors $\rho(F,\mathcal{I})$ est vraie s'il existe $d \in D$ tel que l'interprétation de la formule $G[X \hookleftarrow d]$ soit vraie. Elle est fausse sinon,*
- *si $F = \forall X G$, alors $\rho(F,\mathcal{I})$ est vraie si pour tout $d \in D$, l'interprétation de la formule $G[X \hookleftarrow d]$ est vraie. Elle est fausse sinon.*

Les définitions de modèle, formule valide, formule satisfiable et insatisfiable restent valables, nous les rappelons pour mémoire.

Définition 2.6.5 - Modèle - *Un modèle d'une formule F est une interprétation $\mathcal{I}$ telle que $\rho(F,\mathcal{I}) = t$.*

Définition 2.6.6 - Formule valide - *Une formule F est dite valide si toute interprétation $\mathcal{I}$ de F est un modèle de F.*

Définition 2.6.7 - Formule satisfiable - *Une formule F est dite satisfiable si elle admet au moins un modèle.*

Définition 2.6.8 - Formule insatisfiable - *Une formule F est dite insatisfiable si elle n'admet aucun modèle.*

Nous donnons à présent un exemple d'interprétation pour une formule. Soit la formule :

$$F = \forall X(p(X) \to q(f(X),a)$$

et soit $\mathcal{I}$, l'interprétation définie par :

- un domaine de pré-interprétation $D = \{1,2\}$,
- un fonction d'interprétation des constantes ρ_C telle que :

	a
ρ_C	1

– un fonction d'interprétation des fonctions ρ_F telle que :

	f(1)	f(2)
ρ_F	2	1

– un fonction d'interprétation des atomes ρ_P telle que :

	p(1)	p(2)	q(1,1)	q(1,2)	q(2,1)	q(2,2)
ρ_P	f	t	t	t	f	t

F étant quantifiée universellement, il nous faut, en vertu de la définition 2.6.4, évaluer le sens de F pour tout $d \in D$, soit :

pour $X = 1$

$$\begin{array}{rcl} \rho(F,\mathcal{I}) & = & \rho(\, p(1) \rightarrow q(f(1),a)\,) \\ & = & \rho(\, p(1) \rightarrow q(2,1)\,) \\ & = & \rho(\, f \rightarrow f\,) \\ & = & t \end{array}$$

pour $X = 2$

$$\begin{array}{rcl} \rho(F,\mathcal{I}) & = & \rho(\, p(2) \rightarrow q(f(2),a)\,) \\ & = & \rho(\, p(2) \rightarrow q(1,1)\,) \\ & = & \rho(\, t \rightarrow t\,) \\ & = & t \end{array}$$

F est donc vraie puisqu'elle est vraie pour tout $d \in D$.

2.7 Forme normale

Établir la consistance d'une formule du calcul des prédicats est un problème difficile. On peut alors essayer de travailler sur une version de forme plus simple mais de consistance équivalente à la formule initiale.

Nous introduisons dans cette section la forme *clausale* pour la logique des prédicats. Toute formule de la logique des prédicats du premier ordre admet une représentation sous forme de clause qui préserve sa satisfiabilité. À la différence des formes normales, une clause n'est pas logiquement équivalente à la formule dont elle dérive.

Nous décrivons, à présent, les différentes étapes qui mènent à une représentation sous forme clausale.

Définition 2.7.1 - Forme prénexe - *Une formule F est sous forme prénexe lorsqu'elle a la forme :*

$$\mathcal{Q}_1 X_1\ \mathcal{Q}_2 X_2 \ldots \mathcal{Q}_n X_n\, G$$

où chaque $\mathcal{Q}_i$ est un quantificateur (existentiel ou universel), la partie $\mathcal{Q}_1 X_1\ \mathcal{Q}_2 X_2 \ldots \mathcal{Q}_n X_n$ étant appelée préfixe *et G étant qualifiée de* matrice *ne contient aucun quantificateur.*

Le but d'une formule prénexe est de renvoyer tous les quantificateurs au début de la formule.

Théorème 2.7.1 - Forme prénexe équivalente - *Toute formule admet une forme prénexe équivalente.*

Étant donnée une formule F sous forme prénexe, on applique un procédé, appelé *Skolémisation*, afin de remplacer toutes les variables quantifiées existentiellement par des *fonctions de Skolem* qui permettent d'obtenir une formule F' sans quantificateurs (dans laquelle les variables sont implicitement quantifiées universellement).

Définition 2.7.2 - Algorithme de Skolémisation - *Soit F une formule sous forme prénexe. La formule de Skolem de F s'obtient en :*

1. *remplaçant toute variable quantifiée existentiellement par un symbole de fonction dont les arguments sont des variables qui sont quantifiées universellement et qui précèdent ladite variable,*
2. *supprimant les quantificateurs existentiels, devenus inutiles.*

La formule obtenue par application des règles précédentes est sous forme standard de Skolem.

Exemple : la formule suivante

$$\exists X_1\, \forall X_2\, \forall X_3\, \exists X_4\, \forall X_5\, \exists X_6\, p(X_1{,}X_2{,}X_3{,}X_4{,}X_5{,}X_6)$$

a pour formule de Skolem :

$$\forall X_2\, \forall X_3\, \forall X_5\, p(a_s{,}X_2{,}X_3{,}f_s(X_2{,}X_3){,}X_5{,}g_s(X_2{,}X_3{,}X_5))$$

Dans la formule précédente, $\exists X_1$ n'est précédé par aucun quantificateur universel. C'est pourquoi on introduit une nouvelle constante a_s. $\exists X_4$ est précédé par $\forall X_2$ et $\forall X_3$. On remplace alors X_4 par un nouveau symbole de fonction f_s d'arité 2. De même, $\exists X_6$ est précédé par $\forall X_2$, $\forall X_3$, $\forall X_5$, on introduit la fonction g_s d'arité 3.

Définition 2.7.3 - Forme clausale - *Une formule est sous forme clausale si elle est sous forme standard de Skolem et que sa matrice est sous forme normale conjonctive.*

Théorème 2.7.2 Satisfiabilité d'une clause - *Soit S un ensemble de clauses résultat de la transformation sous forme clausale d'une formule F. F est insatisfiable si et seulement si S est insatisfiable.*

Ce théorème forme la base de nombreux démonstrateurs automatiques utilisant une représentation des formules sous forme de clauses. Il établit que la recherche de l'insatisfiabilité d'une formule F est équivalente à la recherche d'insatisfiabilité de sa représentation sous forme clausale S.

Cependant F et S ne sont pas logiquement équivalentes, mais seule la satisfiabilité est préservée.

2.8 Propriétés fondamentales

Définition 2.8.1 - Adéquation - *Le calcul des prédicats est adéquat.*

Définition 2.8.2 - Consistance - *Le calcul des prédicats est consistant.*

Définition 2.8.3 - Complétude - *Le calcul des prédicats est fortement complet.*

Définition 2.8.4 - Décidabilité - *Le calcul des prédicats est semi-décidable.*

En d'autres termes, il existe une procédure mécanique qui permet d'établir en un temps fini qu'une formule est un théorème. Dans le cas où la formule n'est pas un théorème, la procédure peut ne pas terminer.

Chapitre 3

Démonstration Automatique

« *The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.* »

K. Gödel

Après une présentation des bases de la logique, nous décrivons dans ce chapitre plusieurs procédés de résolution de la logique des prédicats du premier ordre. Cette présentation non exhaustive est motivée par un désir d'embrasser l'étendue des méthodes de preuve de la Démonstration Automatique dans un souci de procurer au lecteur une vision aussi complète que possible de ce domaine de recherche.

3.1 Introduction

Nous allons commencer notre investigation des méthodes de preuve par un rappel des différentes notions sous-jacentes à la Démonstration Automatique.

Définition 3.1.1 - Théorie - *Soit $H = \{H_1, \ldots, H_n\}$ un ensemble de formules. La théorie définie par H, notée $Th(H)$, est l'ensemble des formules F qui découlent logiquement de H ($H \vdash F$). L'ensemble de formules H représente les axiomes de la théorie.*

Les problèmes que l'on cherche à résoudre en Démonstration Automatique s'énoncent alors en ces termes :

Étant donné un ensemble d'axiomes H et une formule F, F est-elle un théorème de $Th(H)$? Autrement dit, a-t-on $H_1 \wedge \ldots \wedge H_n \vdash F$?

Le calcul des prédicats étant adéquat, il résulte de la question précédente que montrer que F est un théorème de $Th(H)$ est équivalent à montrer que F est conséquence logique de H. On a alors recours à une procédure de preuve pour montrer la possible dépendance logique existant entre H et F.

Définition 3.1.2 - Procédure de preuve - *Une procédure de preuve P est un algorithme capable d'établir si une formule F est un théorème d'une théorie H. Si c'est le cas, on dit que F est dérivable de H par P et on note $H \vdash_P F$. Une procédure de preuve accepte en entrée une formule ou un ensemble de clauses et fournit en sortie une réponse binaire du type oui/non indiquant si la formule est ou non un théorème.*

L'expression *méthode de preuve* ou *procédure de preuve* est généralement employée pour décrire une procédure de décision dans le cas semi-décidable. Nous devons ici insister sur la semi-décidabilité du calcul des prédicats qui est de nature à éclairer la difficulté de traitement des problèmes de la logique du premier ordre. La semi-décidabilité impose que pour chaque procédure de preuve du calcul des prédicats, il existe une formule satisfiable, pour laquelle la procédure ne termine pas. En d'autres termes, une procédure de preuve obtiendra une preuve de F si F est effectivement un théorème de $Th(H)$, mais ne terminera pas forcément si on lui demande d'établir une preuve d'un non théorème.

Il semble *a priori* difficile d'établir une classification des procédures de preuve tant les critères de différenciation sont nombreux et variés. Ainsi, on peut distinguer les procédures qui fonctionnent par *affirmation* (tentant de prouver que $H \wedge F$ est valide) ou celles qui fonctionnent par *réfutation* (qui tentent de prouver que $H \wedge \neg F$ est insatisfiable) et qui semblent les plus répandues.

On peut également différencier les procédures traitant des formes clausales de celles qui traitent des formules, ou encore, les procédures qui ont une approche syntaxique et celles qui ont une approche sémantique. Certaines méthodes procèdent par *chaînage avant* (en partant des axiomes pour prouver un théorème) alors que d'autres procèdent par *chaînage arrière* (en partant du théorème pour remonter aux axiomes).

Cela dit, on exige généralement d'une procédure de preuve qu'elle réponde à des propriétés qui assurent sa validité du point de vue de la démonstration. On doit notamment prendre en considération la *terminaison* et la *correction* de la procédure. La *terminaison* détermine dans quelles circonstances l'algorithme s'arrête. Quant à la *correction*, elle assure que l'algorithme termine avec une réponse correcte il suffit pour cela de démontrer l'*adéquation* et la *complétude* de la procédure de preuve.

Définition 3.1.3 - Adéquation d'une procédure de Preuve - *Une procédure de preuve est dite adéquate*[15] *si elle ne peut prouver que des théorèmes.*

Autrement dit, si la procédure de preuve termine avec la réponse *oui*, la formule est obligatoirement un théorème.

Définition 3.1.4 - Complétude d'une procédure de preuve - *Une procédure de preuve P est dite complète si tout théorème (ou formule valide) est prouvable dans P.*

Autrement dit, la procédure de preuve termine toujours avec la réponse *oui* si la formule est un théorème.

Certaines procédures de preuve, comme le *principe de résolution*, font également référence à la notion de consistance.

Définition 3.1.5 - Consistance d'une procédure de preuve - *Une procédure de preuve P est consistante s'il n'existe pas de formule F telle que $\vdash_P F$ et $\vdash_P \neg F$.*

Nous pouvons à présent définir les notions connexes aux procédures de preuve que sont les notions d'espace de recherche, de restriction, de raffinement et d'heuristique.

Définition 3.1.6 - Espace de recherche - *On appelle espace de recherche $\mathcal{G}$, le résultat de l'application des règles d'inférence R, d'une procédure de preuve P, sur un ensemble de formules H. On a alors,*

$$\begin{array}{lcl} \mathcal{G}_0 & = & H \\ \mathcal{G}_i & = & R(\mathcal{G}_{i-1}) \quad i > 0 \\ \mathcal{G} & = & \cup_{i \geq 0} \mathcal{G}_i \end{array}$$

Définition 3.1.7 - Restriction - *On appelle restriction d'une procédure de preuve, une procédure qui génère un sous-ensemble de l'espace de recherche $\mathcal{G}$.*

Par exemple, le recours à des techniques de calcul pondéral sur les littéraux[16] constitue une restriction. Le *weighting* consiste à attribuer une valeur (ou poids) aux symboles en fonction de l'importance qu'on leur accorde dans la recherche de la preuve. Cette technique permet d'éliminer les littéraux dont le poids dépasse une valeur limite fixée par l'utilisateur et sensés être inutiles pour la recherche de la preuve.

Définition 3.1.8 - Stratégie - *On appelle stratégie pour une procédure de preuve, une procédure qui réordonne le développement de l'espace de recherche.*

15. sound en anglais.
16. weighting en anglais.

La recherche en largeur d'abord, en profondeur d'abord ou par approfondissements successifs [61] sont des exemples de stratégies.

Définition 3.1.9 - Raffinement - *Un raffinement d'une procédure de preuve est une restriction et/ou une stratégie.*

Un raffinement est une règle ou un ensemble de règles qui gouvernent l'utilisation des règles d'inférence. Ainsi que nous venons de le voir, on distingue les raffinements *restrictifs* (restriction) qui restreignent l'application des règles d'inférence et les raffinements *directifs* (stratégies) qui dictent l'enchaînement et les conditions d'application des règles d'inférence. Le choix d'un raffinement consiste généralement à imposer la connaissance de l'utilisateur sur le domaine étudié ou à modifier, suivant l'intuition de celui-ci, l'angle d'attaque du problème .

Définition 3.1.10 - Heuristique - *On appelle heuristique un ensemble de règles (généralement empiriques) qui permet de parcourir un sous-espace de l'espace de recherche.*

À vrai dire, les notions de restriction et d'heuristiques sont très proches. Cependant, une heuristique exploite majoritairement des critères sémantiques qui sont sans rapport avec les modes d'applications structurels des règles d'inférence. Un exemple de critère heuristique pourraît être l'application préférentielle des règles d'inférence sur les clauses les plus courtes (en nombres de littéraux).

Le critère de différenciation que nous avons choisi pour classer les procédures de preuves qui suivent, est le critère syntaxe/sémantique. Il semble en effet être le plus à même de rendre compte de la diversité des techniques élaborées afin de résoudre les difficiles problèmes de la Démonstration Automatique. De façon plus précise, nous classons les procédures de preuve en trois catégories :

- les procédures syntaxiques, basées sur une approche de traitement symbolique des formules,
- les procédures sémantiques, applications directes du théorème de Herbrand,
- les procédures syntaxico-sémantiques, combinant les deux aspects précédents.

Nous donnons, pour chacune des approches présentées, son fondement théorique ainsi qu'un exemple explicatif de son fonctionnement. Certains exemples sont donnés en calcul propositionnel pour des raisons évidentes de simplification. Chacune des procédures de preuve présentées est adéquate et complète.

3.2 Méthodes syntaxiques

Dans le cadre de la logique des prédicats, les méthodes syntaxiques prennent tout leur sens. L'existence d'une infinité d'interprétations pour une formule, fait de la sémantique un objet purement théorique. La syntaxe, pourtant lourde et peu efficace dans le cadre du calcul propositionnel, redevient donc attractive à l'ordre supérieur.

3.2.1 Sequent calculus

Gentzen [49] inventa deux procédures de preuve syntaxiques : la *Déduction Naturelle* [95], qui capturait selon ses propres dires le coeur du raisonnement logique, et le *Sequent Calculus* qu'il utilisa comme un outil pratique pour démontrer ses résultats les plus importants.

On peut concevoir le sequent calculus comme une métathéorie développée sur la base d'un langage objet (voir section 1.1.1). Dans cette métathéorie, les lettres grecques Γ,Δ représentent des ensembles de formules, alors que les lettres romaines capitales X,Y représentent une formule.

Les axiomes du sequent calculus sont les suivants :

$$X \vdash X \qquad \bot \vdash \qquad \vdash \top$$

Les règles d'inférences sont :

- **Raffinement**

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ (Raf) \quad \textit{si } \Gamma_1 \subseteq \Gamma_2 \textit{ et } \Delta_1 \subseteq \Delta_2$$

- **Règles de Négation**

$$\frac{\Gamma \vdash \Delta{,}X}{\Gamma{,}\neg X \vdash \Delta}\;(Neg_1) \qquad\qquad \frac{\Gamma{,}X \vdash \Delta}{\Gamma \vdash \Delta{,}\neg X}\;(Neg_2)$$

- **Règles de conjonction**

$$\frac{\Gamma{,}X{,}Y \vdash \Delta}{\Gamma{,}X \wedge Y \vdash \Delta}\;(Conj_1) \qquad\qquad \frac{\Gamma \vdash \Delta{,}X \quad \Gamma \vdash \Delta{,}Y}{\Gamma \vdash \Delta{,}X \wedge Y}\;(Conj_2)$$

- **Règles de disjonction**

$$\frac{\Gamma{,}X \vdash \Delta \quad \Gamma{,}Y \vdash \Delta}{\Gamma{,}X \vee Y \vdash \Delta}\;(Disj_1) \qquad\qquad \frac{\Gamma \vdash \Delta{,}X{,}Y}{\Gamma \vdash \Delta{,}X \vee Y}\;(Disj_2)$$

- **Règles d'implication**

$$\frac{\Gamma \vdash \Delta{,}X \quad \Gamma{,}Y \vdash \Delta}{\Gamma{,}X \to Y \vdash \Delta}\;(Impl_1) \qquad\qquad \frac{\Gamma{,}X \vdash \Delta{,}Y}{\Gamma \vdash \Delta{,}X \to Y}\;(Impl_2)$$

- **Règles** γ $(\forall)$

$$\frac{\Gamma{,}\gamma(t) \vdash \Delta}{\Gamma{,}\gamma \vdash \Delta}\;(\gamma_1) \qquad\qquad \frac{\Gamma \vdash \Delta{,}\gamma(p)}{\Gamma \vdash \Delta{,}\gamma}\;(\gamma_2)$$

- **Règles** δ $(\exists)$

$$\frac{\Gamma \vdash \Delta{,}\delta(t)}{\Gamma \vdash \Delta{,}\delta}\;(\delta_1) \qquad\qquad \frac{\Gamma{,}\delta(p) \vdash \Delta}{\Gamma{,}\delta \vdash \Delta}\;(\delta_2)$$

Pour les règles δ, p est une constante quelconque qui n'apparaît pas précédemment dans la preuve. Pour les règles γ, t symbolise n'importe quelle constante de la preuve.

Exemple La démonstration par chaînage avant de $(\forall X\, p(X) \to q(X)) \to (\forall X\, p(X) \to \forall X\, q(X))$ est la suivante (il faut la lire de bas en haut) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg p(a) \vdash \neg p(a)}{\neg p(a) \vdash q(a),\neg p(a)}\,(Raf)}{\vdash q(a),\neg p(a),p(a)}\,(Neg_2) \qquad \dfrac{q(a) \vdash q(a)}{q(a) \vdash q(a),\neg p(a)}\,(Raf)}{p(a) \to q(a) \vdash q(a),\neg p(a)}\,(Impl_1)}{\forall X\, p(X) \to q(X) \vdash q(a),\neg p(a)}\,(\gamma_1)}{[\forall X\, p(X) \to q(X)], p(a) \vdash q(a)}\,(Neg_1)}{[\forall X\, p(X) \to q(X)], \forall X\, p(X) \vdash q(a)}\,(\gamma_1)}{[\forall X\, p(X) \to q(X)], \forall X\, p(X) \vdash \forall X\, q(X)}\,(\gamma_2)}{[\forall X\, p(X) \to q(X)] \vdash \forall X\, p(X) \to \forall X\, q(X)}\,(Impl_2)}{\vdash [\forall X\, p(X) \to q(X)] \to [\forall X\, p(X) \to \forall X\, q(X)]}\,(Impl_2)$$

3.2.2 Tableaux sémantiques

La méthode des tableaux sémantiques, introduite par Robert **Smullyan** [114], issue des travaux de E. W. **Beth** en 1955, est étroitement liée aux formes normales disjonctives. Dans son principe, elle représente une méthode analytique qui s'articule autour d'une décomposition des formules en formules plus simples, faisant ainsi apparaître la structure logique de la formule initiale, mais en laissant de côté les détails qui ne sont pas essentiels.

Les preuves, dans une telle approche, prennent la forme d'*arbres*, où chaque noeud est étiqueté par une formule. On définit alors des règles de réécriture sur les arbres, appelées *règles d'expansion*, qui consistent à développer les branches d'un arbre pour faire apparaître la structure de la formule à démontrer. Ces règles sont classées en quatre catégories selon l'agencement des formules (voir tables 3.1 et 3.2) :

- les *règles* α pour les formes conjonctives,

– les *règles* β pour les formes disjonctives,
– les *règles* γ pour les formes quantifiées universellement,
– les *règles* δ pour les formes quantifiées existentiellement.

$\dfrac{\neg\neg F}{F}$	$\dfrac{\neg\top}{\bot}$	$\dfrac{\neg\bot}{\top}$	$\dfrac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}}$	$\dfrac{\beta_1 \vee \beta_2}{\beta_1 \vert \beta_2}$	$\dfrac{\beta_1 \rightarrow \beta_2}{\neg\beta_1 \vert \beta_2}$	$\dfrac{\beta_1 \equiv \beta_2}{\begin{array}{cc}\beta_1 & \neg\beta_1\\ \beta_2 & \neg\beta_2\end{array}}$

TAB. 3.1 – *Règles d'expansion α et β*

γ	$\gamma(t)$	δ	$\delta(t)$
$(\forall X\, F[X])$	$F[X \hookleftarrow t]$	$(\exists X\, F[X])$	$F[X \hookleftarrow t]$
$\neg(\exists X\, F[X])$	$\neg F[X \hookleftarrow t]$	$\neg(\forall X\, F[X])$	$\neg F[X \hookleftarrow t]$

TAB. 3.2 – *Règles d'expansion γ et δ*

Définition 3.2.1 - *Tout tableau T^* est obtenu par application des règles d'expansion sur un tableau T précédent.*

Pour démontrer $H_1 \wedge \ldots \wedge H_n \vdash F$, on commence par construire un tableau *à une branche* (voir figure 3.1), puis on applique les règles d'expansion sur chaque branche du tableau jusqu'à obtenir une branche close ou ne plus pouvoir appliquer aucune règle.

$$\begin{array}{c} H_1 \\ \vdots \\ H_n \\ \neg F \end{array}$$

FIG. 3.1 – *Tableau initial à une branche.*

Définition 3.2.2 - Branche close - *Une branche d'un tableau est dite close si F et $\neg F$ apparaissent dans cette branche ou si $\bot$ termine cette branche.*

Définition 3.2.3 - Tableau clos - *Un tableau est clos si toutes ses branches sont closes.*

Théorème 3.2.1 - Théorème d'insatisfiabilité- *Si F est un théorème de $Th(H)$ alors il existe un tableau clos pour F résultant de l'application des règles d'expansion sur un tableau initial de $H \wedge \neg F$.*

Preuve voir [41] p. 56.

L'application des règles d'expansion se fait de manière récurrente sur chaque branche ouverte de l'arbre. L'ordre d'application des règles est en principe indifférent, mais il semble plus intéressant de donner la préférence aux règles qui ont un développement vertical (règles α) plutôt qu'horizontal (règles β) afin d'économiser des distributions sur plusieurs branches (voir figure 3.2).

L'adéquation et la complétude des tableaux sémantiques sont démontrées dans [41] p. 143 et 144.

A titre d'exemple, nous allons mettre en pratique les règles d'expansion sur une instance propositionnelle et une instance du calcul des prédicats.

Exemple Deux démonstrations de $p \to q, q \to s, \neg r \vee \neg s \vdash (\neg p \vee \neg q)$ sont fournies figure 3.2. On distingue clairement que la deuxième démonstration (à droite) est plus "économique" que la première bien que seules les règles β influent ici sur le développement de l'arbre. Nous avons fait figurer en regard de chaque développement de l'arbre, le type de règle appliquée ainsi que la formule sur laquelle elle s'applique.

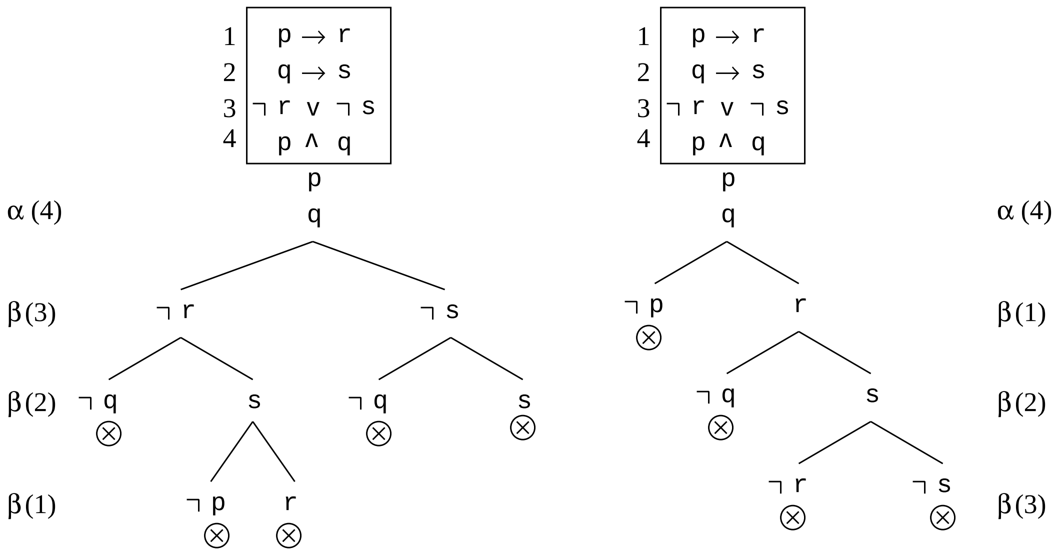

FIG. 3.2 – *Exemple de démonstration avec des tableaux sémantiques en calcul propositionnel.*

Exemple Le théorème suivant est démontré figure 3.3.

$$\neg[(\forall X\, p(X) \vee q(X)) \to ((\exists X\, p(X)) \vee (\forall X\, q(X)))]$$

1	$\neg[(\forall X\, p(X) \vee q(X)) \to (\exists X\, p(X) \vee \forall X\, q(X))]$		
2	$(\forall X\, p(X) \vee q(X))$		$\alpha(1)$
3	$\neg(\exists X\, p(X) \vee \forall X\, q(X))$		
4	$\neg(\exists X\, p(X))$		$\alpha(3)$
5	$\neg(\forall X\, q(X))$		
6	$\neg q(a)$		$\delta(5)$
7	$\neg p(a)$		$\gamma(4)$
8	$p(a) \vee q(a)$		$\gamma(2)$
9	p(a) ⊗	q(a) ⊗	$\beta(8)$

FIG. 3.3 – *Exemple de démonstration avec des tableaux sémantiques en calcul des prédicats.*

3.2.3 Format de Réduction de Problème

Le *Format*[17] *de Réduction de problème*[18] (FRP) est une procédure de preuve déjà ancienne puisqu'elle a été implantée sous une forme spécifique dans la *Logic Theory Machine* de **Newell**, **Simon** et **shaw** [87].

Dans son principe, le format de réduction de problème travaille sur des formules sous forme d'implication $H_1 \wedge \ldots \wedge H_n \to F$. Pour montrer F, il suffit de montrer que chaque H_i se *réduit* à un axiome (voir [88] pour une description approfondie).

Le (PRF) a été repris et adapté par David **Plaisted** [93, 94]. La première version de cette procédure de preuve est le Format de Réduction de Problème Simplifié[19] (FRPS), décrit dans [93], mais dont la règle de *case splitting* est difficile à contrôler si l'on procède par chaînage arrière, ce qui conduit son auteur à introduire une variante du SPRF, appelée Format de Réduction de Problème Modifié[20] (FRPM).

3.2.3.1 Format de Réduction de Problème Simplifié

Cette procédure de preuve se fonde premièrement sur une réécriture des clauses en entrée sous forme de *pseudo clauses de Horn*.

Définition 3.2.4 - But - *Un but ou sous-but est une expression de la forme $\Gamma \to L$ où L est un littéral et $\Gamma = L_1 \wedge \ldots \wedge L_k$, une liste de littéraux appelée hypothèse.*

Définition 3.2.5 - Transformation en pseudo clauses de Horn - *On définit une transformation des clauses en pseudo clauses de Horn. Une pseudo clause de Horn s'articule autour d'une décomposition d'une clause en une tête et un corps (voir table 3.3). Les clauses unitaires composées d'un unique littéral positif restent inchangées.*

Clause	Pseudo clause de Horn
$L \vee \neg L_1 \vee \ldots \vee \neg L_k$	L :-$L_1, \ldots, L_k$
$\neg L_1 \vee \ldots \vee \neg L_k$	$false$:-$L_1, \ldots, L_k$
$M_1 \vee \ldots \vee M_n \vee \neg L_1 \vee \ldots \vee \neg L_k$	$M_1, \ldots, M_n$:-$L_1, \ldots, L_k$

TAB. 3.3 – *Règles de transformation des clauses*

Une fois le problème initial traduit en pseudo clauses de Horn, on peut songer à le transformer en axiomes et règles d'inférence :

– pour tout littéral positif L, on crée l'axiome (ou but) :

$$\Gamma \to L, \quad \Gamma \textit{ est arbitraire et éventuellement vide}$$

– pour toute clause de Horn L :-$L_1, \ldots, L_k$, on définit la règle d'inférence :

$$\frac{\Gamma \to L_1, \ldots, \Gamma \to L_k}{\Gamma \to L}$$

– pour toute clause de la forme $M_1, \ldots, M_n : -L_1, \ldots, L_k$, on introduit une règle dite de *case splitting* (ou de décomposition par cas) :

$$\frac{\Gamma \to L_1, \ldots, \Gamma \to L_k, \Gamma, M_1 \to U, \ldots \Gamma, M_n \to U}{\Gamma \to U}$$

17. le mot format est à prendre au sens de procédure.
18. Problem Reduction Format.
19. *Simplified Problem Reduction Format.*
20. *Modified Problem Reduction Format.*

Exemple Soit le problème propositionnel composé de quatre clauses :

$$\begin{array}{rcr} p & \vee & q \\ p & \vee & \neg q \\ \neg p & \vee & q \\ \neg p & \vee & \neg q \end{array}$$

Les clauses sont traduites en pseudo clauses de Horn :

$$\begin{array}{rcl} p,q & :- & \\ p & :- & q \\ q & :- & p \\ false & :- & p,q \end{array}$$

qui sont ensuite transformées en règles d'inférence :

$$\frac{\Gamma,p \to U, \quad \Gamma,q \to U}{\Gamma \to U}\ (R1)$$

$$\frac{\Gamma \to q}{\Gamma \to p}\ (R2)$$

$$\frac{\Gamma \to p}{\Gamma \to q}\ (R3)$$

$$\frac{\Gamma \to p, \quad \Gamma \to q}{\Gamma \to false}\ (R4)$$

Une preuve de l'ensemble d'axiomes précédent prend alors la forme suivante (ici, Γ et U peuvent être remplacés par n'importe quelle variable propositionnelle) :

$$\dfrac{\dfrac{p \to p \quad \dfrac{p \to p}{p \to q}\,(R3,\Gamma = p)}{p \to false}\,(R4,\Gamma = p) \qquad \dfrac{q \to p \quad \dfrac{q \to q}{q \to q}\,(R2,\Gamma = q)}{q \to false}\,(R4,\Gamma = q)}{\to false}\,(R1,\Gamma = \emptyset,U = false)$$

L'adéquation et la complétude du Format de Réduction de Problème Simplifié sont démontrées dans [94] p. 293.

Le principal inconvénient de ce système réside dans la règle de décomposition par cas (ou *case splitting*), difficile à contrôler si on choisit une approche par chaînage arrière. En effet, cette règle peut être utilisée pour tout Γ et U puisqu'ils n'apparaissent pas dans la clause initiale. Γ et U prenant ici leurs valeurs dans $\{p,q,false,\emptyset\}$. Plaisted propose donc une variante du FRPS, appelée Format de Réduction de Problème Modifié.

3.2.3.2 Format de Réduction de Problème Modifié

Le but du Format de Réduction de Problème Modifié est de diminuer le nombre de cas à examiner lors de l'utilisation de la règle de *case splitting*. Dans cette approche, toutes les clauses sont d'abord traduites en clauses de Horn. On introduit pour cela un nouveau prédicat $not(X)$ et l'on ajoute aux règles d'inférence le schéma d'axiome $not(X) \vee X$. Ainsi, toute clause de la forme

$$M_1 \vee \ldots \vee M_n \vee \neg L_1 \vee \ldots \vee \neg L_k$$

sera traduite en une clause de Horn en choisissant un des littéraux positifs comme tête de la clause, les autres littéraux positifs étant niés (*not*) et ajoutés au corps de la clause de Horn. On choisira donc l'une des représentations suivantes parmi les n possibles :

$$\begin{array}{l} M_1 : -L_1, \ldots, L_k, not(M_2), \ldots, not(M_n) \\ \ldots \\ M_n : -L_1, \ldots, L_k, not(M_1), \ldots, not(M_{n-1}) \end{array}$$

Chaque clause de Horn $L : -L_1, \ldots, L_k$ est ensuite transformée en une règle d'inférence comme suit :

$$\frac{\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1, \quad \Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2, \quad \ldots, \quad \Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

La règle de *case splitting* s'énonce dans ce système :

$$\frac{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, \neg M \rightarrow L, \quad \Gamma_1, M \rightarrow L \Rightarrow \Gamma_1, M \rightarrow L}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L,} \; (Split)$$

On a également recours aux axiomes suivants :

$$\begin{array}{ll} \Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L, & si\ L \in \Gamma\ est\ un\ litt\acute{e}ral \\ \Gamma \rightarrow \neg L \Rightarrow \Gamma, \neg L \rightarrow \neg L, & o\grave{u}\ L\ est\ un\ litt\acute{e}ral\ positif \end{array}$$

Le *Modified Problem Reduction Format* est complet dans le sens où un ensemble de clauses est insatisfiable si et seulement si l'application des règles d'inférence produit la chaîne vide ($\rightarrow false$).

Exemple Revenons à des considérations plus matérielles afin d'exposer le fonctionnement du MPRF sur le problème suivant :

$$\begin{array}{llll} 1 & false & :- & p(X) \\ 2 & p(X) & :- & q(X) \\ 3 & q(X) & :- & r(X) \\ 4 & r(X) & :- & not\, s(X) \\ 5 & p(X) & :- & s(X). \end{array}$$

l'axiomatisation de ce problème donne :

$$\frac{\Gamma_0 \rightarrow p(X) \Rightarrow \Gamma_1 \rightarrow p(X)}{\Gamma_0 \rightarrow false \Rightarrow \Gamma_1 \rightarrow false} \; (R_1)$$

$$\frac{\Gamma_0 \rightarrow q(X) \Rightarrow \Gamma_1 \rightarrow q(X)}{\Gamma_0 \rightarrow p(X) \Rightarrow \Gamma_1 \rightarrow p(X)} \; (R_2)$$

$$\frac{\Gamma_0 \rightarrow r(X) \Rightarrow \Gamma_1 \rightarrow r(X)}{\Gamma_0 \rightarrow q(X) \Rightarrow \Gamma_1 \rightarrow q(X)} \; (R_3)$$

$$\frac{\Gamma_0 \rightarrow \neg s(X) \Rightarrow \Gamma_1 \rightarrow \neg s(X)}{\Gamma_0 \rightarrow r(X) \Rightarrow \Gamma_1 \rightarrow r(X)} \; (R_4)$$

$$\frac{\Gamma_0 \rightarrow s(X) \Rightarrow \Gamma_1 \rightarrow s(X)}{\Gamma_0 \rightarrow p(X) \Rightarrow \Gamma_1 \rightarrow p(X)} \; (R_5)$$

une réfutation pour ce problème est :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\rightarrow \neg s(X) \Rightarrow \neg s(X) \rightarrow \neg s(X)}{\rightarrow r(X) \Rightarrow \neg s(X) \rightarrow r(X)}\,(R_4)}{\rightarrow q(X) \Rightarrow \neg s(X) \rightarrow q(X)}\,(R_3)}{\rightarrow p(X) \Rightarrow \neg s(X) \rightarrow p(X)}\,(R_2) \qquad \dfrac{s(X) \rightarrow s(X) \Rightarrow s(X) \rightarrow s(X)}{s(X) \rightarrow p(X) \Rightarrow s(X) \rightarrow p(X)}\,(R_5)}{\rightarrow p(X) \Rightarrow \rightarrow p(X)}\,(Split)}{\rightarrow false \Rightarrow \rightarrow false}\,(R_1)$$

3.3 Méthodes sémantiques

Les méthodes sémantiques s'établissent sur la base du théorème de Herbrand et consistent à ramener la recherche de satisfiabilité du problème initial à la recherche de satisfiabilité d'instances propositionnelles.

3.3.1 Théorème de Herbrand

Montrer qu'une formule est valide en calcul des prédicats est impossible dans le cas général. Il existe, en effet, une infinité d'interprétations en raison de l'infinité des domaines de pré-interprétation D.

La première étape significative concernant la recherche de méthodes de preuves automatiques pour le calcul des prédicats fut franchie par **Herbrand** [55] en 1930. Une conséquence de son travail fut le fameux théorème de Herbrand, également connu sous l'appellation de théorème de **Skolem-Herbrand-Gödel**. Ce théorème établit qu'un ensemble de *clauses* S est insatisfiable si et seulement si il existe un sous-ensemble fini insatisfiable d'une instance fondamentale de S (voir définition 3.3.4).

Définition 3.3.1 - Univers de Herbrand - *On appelle univers de Herbrand $\mathcal{H}$ d'un ensemble de clauses S, l'ensemble des termes totalement instanciés que l'on peut construire à partir des symboles de constantes C et de fonctions F qui apparaissent dans S. On peut également définir $\mathcal{H}$ de manière inductive :*

$$\begin{array}{lcl} \mathcal{H}_0 & = & C \\ \mathcal{H}_{i+1} & = & \mathcal{H}_i \cup \{f(t_1,\ldots,t_n), \forall f \in F \text{ d'arité } n \text{ et } t_1,\ldots,t_n \in \mathcal{H}_i\} \\ \mathcal{H} & = & \cup_{i\geq 0}\mathcal{H}_i \end{array}$$

Si S ne contient pas de symbole de constante, on ajoute un symbole de constante arbitraire noté a à C. Ainsi, l'univers de Herbrand de la clause $p(f(X,Y)) \vee q(g(X),Z)$ est :

$$\begin{array}{lcl} \mathcal{H}_0 & = & \{a\} \\ \mathcal{H}_1 & = & \mathcal{H}_0 \cup \{g(a),f(a,a)\} \\ \mathcal{H}_2 & = & \mathcal{H}_1 \cup \{g(g(a)),g(f(a,a)),f(a,g(a)),f(g(a),a),f(g(a),g(a)),f(f(a,a),a),\ldots\} \\ \ldots & & \end{array}$$

Définition 3.3.2 - Base de Herbrand - *La base de Herbrand $B_{\mathcal{H}}$ d'un ensemble de clauses S est l'ensemble des atomes totalement instanciés construits à partir des symboles de prédicats de S et des termes de l'univers de Herbrand de S.*

$$\begin{array}{lcl} B^i_{\mathcal{H}} & = & \{p(t_1,\ldots,t_n), \forall p \in P \text{ d'arité } n \text{ et } t_1,\ldots,t_n \in \mathcal{H}_i\} \\ B_{\mathcal{H}} & = & \cup_{i\geq 0}B^i_{\mathcal{H}} \end{array}$$

De retour à l'exemple précédent, cela se traduit par :

$$B_{\mathcal{H}} = \{p(a),p(g(a)),p(f(a,a)),q(a,a),q(a,g(a)),q(g(a),a),\ldots\}$$

Définition 3.3.3 - Interprétation de Herbrand - *Une interprétation de Herbrand $\mathcal{I}_H$ d'un ensemble de clauses S est une interprétation comportant les caractéristiques suivantes :*

- *l'univers de pré-interprétation D est l'univers de Herbrand $\mathcal{H}$ de S,*
- $\forall c \in C, \rho_C(c) = c$,
- $\forall f \in F, \rho_F(f(t_1,\ldots,t_n)) = f(t_1,\ldots,t_n)$,
- ρ_P *est quelconque.*

Une interprétation de Herbrand est bien une interprétation au sens de la définition 2.6.3. Cependant toute interprétation n'est pas toujours une interprétation de Herbrand.

Définition 3.3.4 - Instance fondamentale - *On appelle instance fondamentale d'une clause (resp. d'un ensemble de clauses), une clause (resp. un ensemble de clauses) ne comportant aucun symbole de variable.*

Nous jonglerons parfois avec la terminologie en parlant de clause ou littéral totalement instancié, ou d'instance fondamentale d'une clause ou d'un littéral, étant tacitement entendu que ces notions sont équivalentes.

Définition 3.3.5 - Modèle de Herbrand - *On appelle modèle de Herbdand d'un ensemble de clauses S, une interprétation de Herbrand $\mathcal{I}_H$ qui est un modèle de S.*

Théorème 3.3.1 - Théorème de Skolem-Herbrand-Gödel - *Un ensemble de clauses S est insatisfiable si et seulement si il existe un sous-ensemble fini insatisfiable d'une instance fondamentale de S. En d'autres termes, un ensemble de clauses est insatisfiable si et seulement si il ne possède aucun modèle de Herbrand.*

Preuve voir [71] p. 49 ou [23] p. 55.

En fait, le théorème de Herbrand a pour but de réduire le nombre d'interprétations à examiner pour établir l'insatisfiabilité d'un ensemble de clauses. L'idée essentielle de Herbrand est d'introduire un domaine d'interprétation qui soit un modèle universel et abstrait, sur la base duquel il suffit de travailler afin d'établir la consistance d'une formule dans le cas général.

Le principal défaut du théorème de Herbrand réside dans l'énumération de la base de Herbrand et dans la génération des instances fondamentales du problème considéré. Un procédé simple d'énumération consiste à énumérer la base suivant l'ordre croissant des longueurs des atomes de la base (lorsqu'ils sont écrits sous forme de chaînes de caractères). Par exemple, si $C = \{a,b\}$, $F = \{f,g\}$, $P = \{p,q\}$, (avec f,g,p,q d'arités respectives 2,1,1,2), l'énumération de la base de Herbrand est :

$$\{p(a),p(b),q(a,a),q(a,b),q(b,a),q(b,b),p(g(a)),p(g(b)),p(f(a,a)),\ldots\}$$

L'atome $q(f(a,b),g(a))$ peut être considéré comme une chaîne "$qfabga$" de longueur 6.

Les procédures de preuve basées sur le théorème de Herbrand sont probablement les plus simples que l'on puisse concevoir. Elles consistent à générer pour chaque $\mathcal{H}_i$ l'ensemble S_i de toutes les instances fondamentales de S, obtenues par substitution des variables par des termes de $\mathcal{H}_i$. Il reste alors à vérifier l'insatisfiabilité de chaque $s \in S_i$ par le biais d'un démonstrateur automatique pour le calcul propositionnel.

Gilmore [50] fut le premier à mettre en oeuvre le théorème de Herbrand en 1960. Pour déterminer l'insatisfiabilité d'une instance $s \in S_i$, il utilisait la méthode dite de *multiplication* qui consiste à mettre s sous forme normale disjonctive afin d'en déterminer l'insatisfiabilité (cf. 1.5). Cependant, la méthode de preuve esquissée par Herbrand s'avère assez difficile à implanter et se révèle inefficace. Par exemple, pour le problème suivant tiré de [23] p. 70 :

$$S = \{p(X,g(X),Y,h(X,Y),Z,k(X,Y,Z)), \neg p(U,V,e(V),W,f(V,W),X))\}$$

l'univers de Herbrand de ce problème est successivement :

$$\begin{aligned}
&\mathcal{H}_0 = \{a\} \\
&\mathcal{H}_1 = \{a,g(a),e(a),h(a,a),f(a,a),k(a,a,a)\} \\
&\ldots
\end{aligned}$$

La base de Herbrand $B_{\mathcal{H}}$ de ce problème comporte 2 littéraux pour $i = 0$, et 1512 ($= 6^3 + 6^4$) pour $i = 1$, ce qui représente $(6^3 \times 6^4)^2$ ensembles de clauses fondamentales différents à tester. Le premier cas d'insatisfiabilité apparaît pour $\mathcal{H}_5$, dont l'ordre de grandeur est de 10^{64}. Il existe donc environ $(10^{64})^2 = 10^{128}$ instances fondamentales à tester pour $i = 5$. La technologie actuelle ne permet même pas de stocker tous les termes de la base de Herbrand de ce problème pour $i = 5$. Notons, en prévision de ce que nous verrons plus tard, que le *principe de résolution* solutionne ce problème instantanément.

Exemple Soit le problème suivant :

$$\begin{array}{l} p(X,f(X)) \\ \neg p(a,X) \vee q(X) \\ \neg q(X) \end{array}$$

pour $\mathcal{H}_0 = \{a\}$, il existe une seule instance fondamentale du problème :

$$\begin{array}{l} p(a,f(a)) \\ \neg p(a,a) \vee q(a) \\ \neg q(a) \end{array}$$

pour $\mathcal{H}_1 = \{a,f(a)\}$, il existe 8 instances fondamentales du problème :

$p(a,f(a))$ $\neg p(a,a) \vee q(a)$ $\neg q(a)$	$p(a,f(a))$ $\neg p(a,a) \vee q(a)$ $\neg q(f(a))$	$p(a,f(a))$ $\neg p(a,f(a)) \vee q(f(a))$ $\neg q(f(a))$	$p(a,f(a))$ $\neg p(a,f(a)) \vee q(f(a))$ $\neg q(a)$
$p(f(a),f(f(a)))$ $\neg p(a,a) \vee q(a)$ $\neg q(a)$	$p(f(a),f(f(a)))$ $\neg p(a,a) \vee q(a)$ $\neg q(f(a))$	$p(f(a),f(f(a)))$ $\neg p(a,f(a)) \vee q(f(a))$ $\neg q(f(a))$	$p(f(a),f(f(a)))$ $\neg p(a,f(a)) \vee q(f(a))$ $\neg q(a)$

dont seule la troisième ($1^{ère}$ ligne, $3^{ème}$ colonne) est insatisfiable (voir également figure 3.4).

3.3.2 Arbres sémantiques

Les arbres sémantiques représentent un procédé plus efficace de recherche d'un sous-ensemble de clauses insatisfiable sur la base de Herbrand que ne l'est l'énumération sur cette base. Ils font référence à la notion d'arbre de recherche. Une branche d'un arbre sémantique représente une interprétation partielle.

Définition 3.3.6 - Littéraux complémentaires - *Si A est un atome, alors A et $\neg A$ sont des littéraux complémentaires.*

Définition 3.3.7 - Arbre sémantique - *Un arbre sémantique est un arbre binaire dont chaque noeud représente une interprétation partielle d'une formule. Deux branches issues d'un même noeud sont étiquetées par des littéraux complémentaires (voir figure 3.4).*

Définition 3.3.8 - Arbre sémantique complet - *Un arbre sémantique est dit complet pour n paires de littéraux complémentaires s'il comporte 2^n branches.*

Notre intérêt n'est pas de construire un arbre sémantique complet pour le problème à résoudre, mais au contraire d'élaguer l'arbre le plus rapidement possible, afin d'éviter des recherches redondantes comme c'est souvent le cas avec les méthodes découlant directement du théorème de Herbrand.

Définition 3.3.9 - Noeud impasse - *Un noeud est qualifié de noeud impasse (ou noeud échec) s'il falsifie une instance d'une clause.*

Définition 3.3.10 - Arbre sémantique clos - *Un arbre sémantique est dit clos si chaque branche de l'arbre se termine par un noeud impasse.*

Théorème 3.3.2 - Théorème d'insatisfiabilité - *Un ensemble de clauses totalement instanciées est insatisfiable si et seulement si il possède un arbre sémantique clos.*

Preuve voir [23] p. 61.

Exemple Reprenons le problème précédent :

$$\begin{array}{lcl} C_1 & = & p(X,f(X)) \\ C_2 & = & \neg p(a,X) \vee q(X) \\ C_3 & = & \neg q(X) \end{array}$$

Une énumération de la base de Herbrand de ce problème est :

$$B_{\mathcal{H}} = \{q(a),p(a,a),q(f(a)),p(a,f(a)),q(f(f(a))),\ldots\}$$

Ce problème est insatisfiable car il possède un arbre clos (voir figure 3.4) sur lequel nous avons fait apparaître les numéros des clauses falsifiées par l'interprétation partielle en cours.

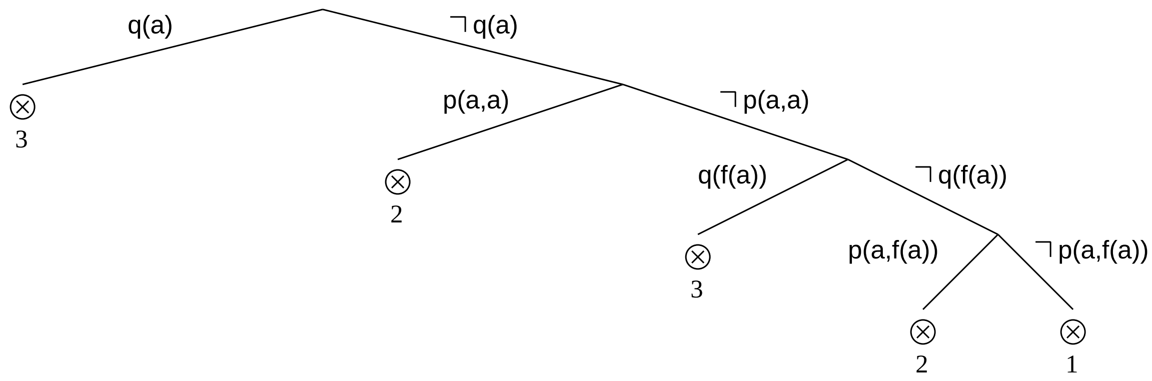

FIG. 3.4 – *Exemple d'arbre sémantique clos.*

3.3.3 Davis et Putnam

La procédure de **Davis** et **Putnam** a été originellement conçue pour le calcul propositionnel. En calcul des prédicats elle s'applique sur un ensemble de clauses S totalement instancié. Le principe de ses règles et de décomposer le problème initial en sous-problèmes plus simples en fonction de la forme des clauses. Elle est constituée de quatre règles :

I. Règle de Tautologie Supprimer de S toutes les clauses qui sont des tautologies,

II. Règle Unitaire S'il existe une clause unitaire L dans S alors construire S_L à partir de S en supprimant toute clause contenant L. Si S_L est vide alors S est satisfiable. Sinon, construire $S_{\neg L}$ obtenu à partir de S_L en supprimant toute occurrence de $\neg L$. $S_{\neg L}$ est insatisfiable si et seulement si S est insatisfiable.

III. Règle du Littéral Pur Un littéral L est dit *pur* pour un ensemble de clauses S, si son complémentaire n'apparaît pas dans S. Dans ce cas, construire $S_{\neg L}$ en supprimant toutes les clauses de S contenant L. $S_{\neg L}$ est insatisfiable si et seulement si S est insatisfiable.

IV. Règle de Décomposition Si l'ensemble S peut être mis sous la forme

$$(A_1 \vee L) \wedge \ldots \wedge (A_m \vee L) \wedge (B_1 \vee \neg L) \wedge \ldots \wedge (B_n \vee \neg L) \wedge R$$

où les A_i, B_i et R ne contiennent pas L et $\neg L$ alors construire les ensembles :

$$\begin{array}{lcl} S_L & = & A_1 \wedge \ldots A_m \wedge R \\ S_{\neg L} & = & B_1 \wedge \ldots B_n \wedge R \end{array}$$

S est insatisfiable si et seulement si S_L et $S_{\neg L}$ sont insatisfiables.

Exemple La résolution du problème suivant est représentée figure 3.5 :

$$\begin{array}{rcl} \neg p & \vee & p \\ \neg p & \vee & q \\ \neg q & \vee & r \\ p & \vee & q \\ \neg r & & \end{array}$$

$\neg p \vee p$
$\neg p \vee q$
$\neg q \vee r$
$p \vee q$
$\neg r$

I p

$\neg p \vee q$
$\neg q \vee r$
$p \vee q$
$\neg r$

II $\neg r$

$\neg p \vee q$
$\neg q$
$p \vee q$

II $\neg q$

$\neg p$
p

II p

$\otimes$

FIG. 3.5 – *Démonstration suivant Davis et Putnam*

3.4 Méthodes syntaxico-sémantiques

Les méthodes de preuve que nous qualifions de syntaxico-sémantiques peuvent être considérées comme des méthodes sémantiques qui se fondent sur un traitement symbolique des termes et s'appuient sur une représentation sous forme clausale des formules.

3.4.1 Principe de résolution

Le principe de résolution, introduit par **Robinson** [104] en 1965, représente la seconde percée majeure dans le domaine de la Démonstration Automatique. L'idée qui fonde le principe de résolution est la suivante : soient deux clauses du calcul propositionnel $C_1 = C_1' \vee L_1$ et $C_2 = C_2' \vee L_2$, où L_1,L_2 sont deux littéraux complémentaires ($L_1 = \neg L_2$) et C_1',C_2' sont des disjonctions de littéraux (éventuellement vides), alors la clause $C_3 = C_1' \vee C_2'$ est conséquence logique de C_1 et C_2. Autrement dit, on a :

$$C_1' \vee L_1, C_2' \vee \neg L_1 \models C_1' \vee C_2'$$

En calcul des prédicats, les littéraux sont de la forme $p(t)$ où p est un symbole de prédicat et t une liste de termes. Dire que deux littéraux complémentaires $L_1 = p(t_1)$ et $L_2 = \neg p(t_2)$ sont égaux revient donc à résoudre l'équation $t_1 = t_2$ sur les termes en substituant, comme il est d'usage en mathématiques, les variables par des sous-termes. L'unification, introduite également par **Robinson**, représente un tel procédé de résolution d'équations sur les termes.

Nous passons à présent à l'examen de l'unification puis présentons le principe de résolution.

Définition 3.4.1 - Unificateur - *Un unificateur d'un ensemble de clauses $C_1, \ldots, C_n$ est une substitution σ telle que $\sigma(C_1) = \ldots = \sigma(C_n)$. On dit alors que $C_1, \ldots, C_n$ sont unifiables.*

Définition 3.4.2 - Plus grand unificateur - *Un unificateur σ d'un ensemble de clauses $C_1,\ldots,C_n$ est un plus grand unificateur, noté mgu*[21]*, si et seulement si pour chaque unificateur θ de l'ensemble il existe une substitution λ telle que $\theta = \sigma\lambda$.*

Définition 3.4.3 - Ensemble de discordance - *L'ensemble de discordance d'un ensemble de littéraux $L_1,\ldots,L_n$ est l'ensemble des sous-termes qui débutent après le dernier symbole commun dans un parcours de gauche à droite des termes des littéraux.*

Ainsi, pour $L_1 = p(a,g(X),b)$ et $L_2 = \neg p(a,g(h(Y)),b)$, l'ensemble de discordance $D(L_1,L_2) = \{X,h(Y)\}$.

Nous pouvons à présent donner l'algorithme d'unification de Robinson où W est l'ensemble des littéraux à unifier et σ_k le plus grand unificateur, si celui-ci existe.

1. $k = 0, W_k = W, \sigma_k = \emptyset$.
2. si W_k est un singleton alors FIN (σ_k est l'unificateur le plus général de W), sinon calculer l'ensemble de discordance D_k de W_k.
3. s'il existe des variables X_k et des termes t_k dans D_k tels que X_k n'apparaisse pas dans t_k alors aller en 4, sinon FIN (l'ensemble n'est pas unifiable).
4. $$\begin{aligned} \sigma_{k+1} &= \sigma_k \circ \{X_k \hookleftarrow t_k\} \\ W_{k+1} &= W_k[X_k \hookleftarrow t_k] \end{aligned}$$
5. $k = k + 1$, aller en 2.

Théorème 3.4.1 - Théorème d'unification - *Si W est un ensemble fini de littéraux unifiables, l'algorithme d'unification terminera toujours et le dernier σ_k représente le plus grand unificateur de W.*

Preuve voir [23] p. 79, l'algorithme termine également si les littéraux ne sont pas unifiables.

L'algorithme d'unification de Robinson présente l'avantage d'être simple mais est relativement peu optimisé. Il existe d'autres algorithmes d'unification de complexité en temps quasi-linéaire comme celui de **Huet** [56], celui de **Martelli - Montanari** [77] ou celui de **Paterson-Wegman** [92]. Pour une présentation et une discussion concernant ces divers algorithmes nous renvoyons le lecteur à [39].

L'étape 3 de l'algorithme, qui consiste à vérifier qu'on ne réalise pas une unification entre une variable et un terme contenant cette variable, s'appelle *test d'occurrence*[22]. Il assure que l'on ne rentrera pas dans une boucle infinie lors d'une substitution. Par exemple, si on unifiait X avec $f(X)$, lors d'une substitution ultérieure de la variable X dans un terme, X serait remplacée par $f(X)$, mais le terme $f(X)$ contenant X serait alors remplacé par $f(f(X))$, etc.

Exemple Voici quelques cas triviaux d'unification possibles ou impossibles :

1. $p(X,b)$ et $p(a,Y)$ sont unifiables, $\sigma_k = \{X \hookleftarrow a, Y \hookleftarrow b\}$,
2. $p(X,X)$ et $p(a,b)$ ne sont pas unifiables, car une fois la substitution $X \hookleftarrow a$ réalisée, les littéraux sont respectivement $p(a,a)$ et $p(a,b)$ qui ne peuvent être égaux,
3. $p(X,X)$ et $p(a,Y)$ sont unifiables, $\sigma_k = \{X \hookleftarrow a, Y \hookleftarrow b\}$,
4. $p(X,f(X))$ et $p(Y,Y)$ ne sont pas unifiables, cas de boucle infinie évoqué précédemment.

À présent que la notion d'unification est fixée, nous en venons à définir le principe de résolution.

Définition 3.4.4 - Résolvante binaire - *Soient $C_1 = C_1' \vee L_1$ et $C_2 = C_2' \vee L_2$ deux clauses (dites clauses parentes) telles que L_1 et L_2 soient des littéraux complémentaires unifiables suivant un plus grand unificateur σ, alors on appelle résolvante binaire de C_1 et C_2, que l'on note $Res(C_1,C_2)$, la clause $C = \sigma(C_1') \vee \sigma(C_2')$. En d'autres termes, on a :*

$$\frac{C_1' \vee L_1 \quad C_2' \vee L_2}{\sigma(C_1') \vee \sigma(C_2')} \ (Res)$$

21. pour most general unifier
22. occurs check en anglais.

Définition 3.4.5 - Principe de résolution - *Le principe de résolution consiste, pour démontrer l'insatisfiabilité d'un ensemble de clauses S, à générer $R(S)$ l'ensemble des résolvantes de S et en ajoutant toute nouvelle résolvante r obtenue à S.*

Définition 3.4.6 - Résolution par saturation - *La méthode de saturation par niveaux*[23] *consiste à générer par couches les résolvantes d'un ensemble de clauses S. En somme, nous avons :*

$$\begin{array}{lcl} S_0 & = & S \\ S_{i+1} & = & S_i \cup R(S_i),\ i \geq 0 \end{array}$$

Théorème 3.4.2 - Théorème d'insatisfiabilité - *Un ensemble de clauses S est insatisfiable si et seulement si il existe une résolvante vide de S (notée $\square$).*

Exemple Démontrons que l'ensemble de clauses suivant est insatisfiable :

$$(P_1)\left\{\begin{array}{ll} C_1 = & \neg p(X) \vee q(X) \\ C_2 = & \neg p(X) \vee r(X) \\ C_3 = & p(a) \\ C_4 = & s(X) \\ C_5 = & \neg s(X) \vee \neg r(X) \end{array}\right.$$

En appliquant la méthode de saturation, on obtient :

$$\begin{array}{l} S_0\left\{\begin{array}{lll} C_1 = & \neg p(X) \vee q(X) & \\ C_2 = & \neg p(X) \vee r(X) & \\ C_3 = & p(a) & \\ C_4 = & s(X) & \\ C_5 = & \neg s(X) \vee \neg r(X) & \end{array}\right. \\ S_1\left\{\begin{array}{lll} C_6 = & q(a) & Res(C_1,C_3) \\ C_7 = & r(a) & Res(C_2,C_3) \\ C_8 = & \neg p(X) \vee \neg s(X) & Res(C_2,C_5) \\ C_9 = & \neg r(X) & Res(C_4,C_5) \end{array}\right. \\ S_2\left\{\begin{array}{lll} C_{10} = & \neg p(X) & Res(C_2,C_9) \\ C_{11} = & \neg s(a) & Res(C_3,C_8) \\ C_{12} = & \neg p(X) & Res(C_4,C_8) \\ C_{13} = & \neg s(a) & Res(C_5,C_7) \\ C_{14} = & \square & Res(C_7,C_9) \end{array}\right. \end{array}$$

L'intérêt du principe de résolution est double. D'une part, nous travaillons sur des littéraux *génériques* et non pas sur des littéraux totalement instanciés. D'autre part, nous ne sommes pas contraints d'utiliser toutes les clauses du problème pour démontrer l'insatisfiabilité du problème, alors que dans le cas du théorème de Herbrand, on est contraint, au minimum, d'instancier toutes les clauses du problème. Ainsi, une réfutation du problème précédent sans saturation et sans faire usage de la clause C_1 est :

$$\begin{array}{lll} C'_6 = & r(a) & Res(C_2,C_3) \\ C'_7 = & \neg r(X) & Res(C_4,C_5) \\ C'_8 = & \square & Res(C_6,C_7) \end{array}$$

Dans l'exemple précédent, nous avons utilisé une règle d'inférence appelée *résolution binaire*. Celle-ci, employée seule, ne saurait suffire à traiter certains problèmes, car elle est incomplète. Ainsi, le problème suivant :

$$\begin{array}{ll} C_1 = & \neg p(X) \vee \neg p(a) \\ C_2 = & p(X) \vee p(Y) \end{array}$$

23. level saturation method en anglais.

bien qu'insatisfiable du point de vue du théorème de Herbrand, ne l'est pas pour la résolution binaire car on ne peut produire la clause vide (chaque résolvante contient toujours 2 littéraux).

$$\begin{array}{lll} C_3 = & \neg p(a) \vee p(Y) & Res(C_1,C_2) \text{ entre } p(X) \text{ et } \neg p(X) \\ C_4 = & \neg p(a) \vee p(X) & Res(C_1,C_2) \text{ entre } p(Y) \text{ et } \neg p(X) \\ C_5 = & \neg p(X) \vee p(Y) & Res(C_1,C_2) \text{ entre } p(X) \text{ et } \neg p(a) \\ C_6 = & \neg p(X) \vee p(X) & Res(C_1,C_2) \text{ entre } p(Y) \text{ et } \neg p(a) \\ \ldots & & \end{array}$$

On introduit la notion de clause factorisée qui permet d'obtenir une règle d'inférence complète.

Définition 3.4.7 - Clause factorisée - *Si plusieurs littéraux de même signe d'une clause C ont un plus grand unificateur σ, alors $\sigma(C)$ est appelée clause factorisée de C.*

Définition 3.4.8 - Clause résolvante - *Une clause résolvante de deux clauses C_1 et C_2 est*

- *soit une résolvante binaire de C_1 et C_2,*
- *soit une résolvante binaire de C_1 et d'une factorisée de C_2,*
- *soit une résolvante binaire d'une factorisée de C_1 et de C_2,*
- *soit une résolvante binaire d'une factorisée de C_1 et d'une factorisée de C_2.*

Le principe de résolution basé sur la notion de résolvante factorisée est complet. Sur l'exemple précédent, la factorisée de C_1 est $fact(C_1) = \neg p(a)$ et la factorisée de C_2 est $fact(C_2) = p(X)$. L'application de la résolution binaire entre ces deux factorisées génère la clause vide.

Le principe de résolution, bien que plus efficace que le théorème de Herbrand, souffre de deux défauts. Le premier concerne la génération de résolvantes *redondantes*. Le second, est la génération de résolvantes *inutiles* pour le but à résoudre. Ces deux défauts peuvent être clairement identifiés lors de la résolution par saturation du problème (P_1). Par exemple, la redondance est illustrée par les couples (C_{10},C_{12}) et (C_{11},C_{13}) pour lesquels on a pu produire deux fois la même résolvante avec des clauses parentes différentes.

La génération de clauses inutiles est également flagrante. Pour obtenir la clause vide C_{14}, il suffit de produire les résolvantes C_7,C_9,C_{14} :

$$\frac{\dfrac{C_2 \quad C_3}{C_7} \quad \dfrac{C_4 \quad C_5}{C_9}}{C_{14}}$$

les clauses C_6,C_8 et C_{10} à C_{13} sont donc générées "inutilement". Nous devons donc songer à élaborer des raffinements du principe de résolution qui ont pour objectif de pallier aux défauts de ce dernier.

Citons pour information la résolution verrouillée[24] [15], la résolution à préférence unitaire[25] [131], la résolution avec ensemble support[26] [133], la résolution sémantique[27] [110], la résolution linéaire[28], [69, 72]. Pour une discussion plus approfondie sur l'ensemble de ces raffinements consulter [23, 71].

Dans une perspective plus large, nous voudrions insister sur les raffinements suivants car ils nous intéressent dans le cadre de nos travaux.

- **Stratégie de l'Ensemble Support**
  La stratégie de l'ensemble support[29] [23] p. 110, est en fait un raffinement qui consiste à réaliser une partition de l'ensemble initial de clauses, en deux sous-ensembles, dont l'un, appelé ensemble support T est composé de clauses satisfiables. On choisit d'ordinaire de constituer T par des clauses puisées dans l'ensemble des axiomes, ces derniers étant généralement satisfiables. On interdit alors l'application du principe de résolution sur deux clauses de T, ce qui a pour conséquence d'éliminer un grand nombre de résolvantes inutiles. La stratégie de l'ensemble support est complète pour la réfutation.

24. lock resolution.
25. unit preference strategy.
26. set of support strategy.
27. semantic resolution.
28. linear resolution.
29. set of support strategy.

- **Résolution unitaire et unit-resulting resolution**
  L'objet de la *résolution unitaire* est de produire une nouvelle résolvante à partir de deux clauses dont l'une est une clause unitaire. L'*unit-resulting resolution* (UR-resolution) consiste à produire une nouvelle résolvante unitaire à partir d'une clause non unitaire et d'un ensemble de clauses unitaires. Elle correspond à une métarègle, résultat de l'application d'une série de résolutions unitaires. Les résolvantes obtenues sont susceptibles de prendre part à de futures applications de la règle d'UR-resolution.

$$\frac{L_1 \vee \ldots \vee L_n \qquad M_1 \quad \ldots \quad M_{n-1}}{\sigma(L_n)}\ (UR)$$

  où σ est un plus grand unificateur des $(M_i,L_i),\forall i,1 \leq i \leq n-1$.
  Cette règle d'inférence n'est complète que pour les ensembles de clauses composés uniquement de clauses de Horn. Elle est néanmoins efficace. « *However efficiency is also important in mechanical theorem proving. Sometimes, we may like to trade completeness for efficiency. That is, there may be some refinements of resolution that are efficient but incomplete. If a refinement of resolution is efficient and powerful enough to prove a large class of therorems, even though it is not complete, it may still be useful.* » [23] p. 133.
  On peut également remarquer que la règle d'UR-resolution s'accorde avec le principe des méthodes sémantiques ou de la génération de modèle (cf. section 3.4.4).
- **Hyperrésolution**
  Le principe de l'*hyperrésolution* [103, 90] découle de la stratégie de l'ensemble support et réside dans la séparation des clauses en deux ensembles : l'ensemble des clauses ne contenant que des littéraux positifs, appelé ensemble des *satellites*, et l'ensemble des clauses contenant au moins un littéral négatif, appelé ensemble *noyau*. L'hyperrésolution s'applique entre une clause du noyau et un ensemble de clauses satellites (autant que de littéraux négatifs dans la clause noyau). Le résultat est une résolvante positive appelée *hyperrésolvante*, ajoutée à l'ensemble des clauses satellites. L'hyperrésolution est un peu moins performante que la résolution unitaire mais offre l'avantage d'être complète.

$$\frac{\neg L_1 \vee \ldots \vee \neg L_n \vee R \qquad M_1 \vee R_1 \quad \ldots \quad M_n \vee R_n}{\sigma(R \vee R_1 \vee \ldots \vee R_n)}\ (HR)$$

  où σ est un plus grand unificateur des $(L_i,M_i),\forall 1 \leq i \leq n$. Les lettres $R,R_1,\ldots,R_n$ sont des disjonctions de littéraux.

3.4.2 Élimination de modèle

L'élimination de modèle[30] [71] est une variante du principe de résolution, axée sur une modélisation des clauses en tant que *chaînes* et une décomposition de la règle de résolution en deux sous-règles.

Définition 3.4.9 - Chaîne - *Une chaîne est une clause de la forme $B_l[A_1]\ldots[A_k]B_r$ où B_l et B_r sont appelés B-littéraux et $A_1,\ldots,A_k$ A-littéraux.*

Le symbole $\vee$, présent dans les clauses, n'apparaît pas dans une chaîne afin d'éviter les écritures trop encombrantes.

L'élimination de modèle utilise trois règles d'inférences :

- la règle d'*extension*, similaire au principe de résolution, à la différence qu'un des littéraux résolu est gardé, promu au rang de A-littéral (un A-littéral étant considéré comme une sorte de mécanisme de mémorisation des règles d'extension successivement appliquées sur une chaîne),
- la règle de *réduction* réalise l'élimination des A-littéraux qui deviennent des *lemmes* et qui représentent des preuves avérées de certains faits.

30. Model Elimination en anglais. L'expression élimination de modèle n'est pas à prendre au sens propre.

– la règle de contraction.

Définition 3.4.10 - Règle d'extension - *Soit C_1 une chaîne de la forme $C_1 = B_l[A_1]\ldots[A_k]B_r l_1$ où l_1 est le littéral le plus à droite de B_r. Soit également une clause C_2 de la forme $l_2 \vee C'_2$ contenant un littéral l_2 de signe complémentaire à l_1 et unifiable avec l_1 suivant le mgu σ. Le résultat de l'application de la règle d'extension entre C_1 et C_2, notée $ext(C_1,C_2,l_2)$, produit la chaîne :*

$$\sigma(\, B_l[A_1]\ldots[A_k][l_1]B_r C'_2\,)$$

Définition 3.4.11 - Règle de réduction - *Soit C_1 une chaîne de la forme $C_1 = B_l[A_1]\ldots[A_k]B_r l_1$ telle qu'il existe un A-littéral A_i $(1 \leq i \leq k)$ de signe complémentaire à l_1 et unifiable avec l_1 suivant le mgu σ.Le résultat de l'application de la règle de réduction sur C_1, noté $red(C_1,A_i)$ produit la chaîne :*

$$\sigma(\, B_l[A_1]\ldots[A_{i-1}]B_r[A_{i+1}]\ldots[A_k]\,)$$

A ces deux règles d'inférence on doit adjoindre une règle, dite de contraction, qui est utilisée après chaque application de la règle d'extension ou de la règle de réduction.

Définition 3.4.12 - Règle de contraction - *Si après extension ou réduction, la sous-chaîne B_r est vide alors les A-littéraux sont supprimés de la chaîne, puis niés et considérés comme prouvés. On leur donne alors le nom de* lemme *ou solution cache.*

Ainsi, si $C_1 = p(X,Z)[r(Y,Z)]q(f(X),Y)$ et $C_2 = \neg q(f(a),c) \vee \neg r(c,b)$, alors

$ext(C_1,C_2,\neg q(f(a),c))$	$=$	$p(a,Z)[r(c,Z)][q(f(a),c)]\neg r(c,b)$	
$red(C_1,r(c,Z))$	$=$	$p(a,b)[r(c,b)][q(f(a),c)]$	sans contraction
$red(C_1,r(c,Z))$	$=$	$p(a,b)$	avec contraction

Cependant tous les A-littéraux ne sont pas forcément promus au rang de lemme.

Définition 3.4.13 - Éligibilité des A-littéraux en tant que lemmes - *Soit $B_l[A_1]\ldots[A_k]B_r$ une chaîne. L'opération de contraction élit A_k au rang de lemme si et seulement si aucun A-littéral A_1 à A_{k-1} n'a été utilisé dans une opération de réduction lors de la formation de la chaîne.*

L'adéquation et la complétude de l'élimination de modèle sont démontrées dans [71] p. 180 et 185.

Exemple Soit l'ensemble d'axiomes suivant :

1	$p(X)\,\neg q(Y)$
2	$\neg p(X)\,r(Y)$
3	$\neg p(a)\,\neg r(Y)$
4	$p(a)\,q(X)$

On choisit l'axiome 4 comme axiome de départ de la démonstration (voir table 3.4).

5	$p(a)[q(Y)]p(X)$	$ext(4,1,\neg q(Y))$
6	$p(a)[q(Y)][p(X)]r(Z)$	$ext(5,2,\neg p(X))$ (avec renommage)
7	$p(a)[q(Y)][p(X)][r(Z)]\neg p(a)$	$ext(6,3,\neg r(Y))$
8	$p(a)$	$red(7,p(X))$ et contraction
	$\neg p(a)$	lemme
	$\neg q(Y)$	lemme
9	$[p(a)]\neg r(Y)$	$ext(8,3,\neg p(a))$
10	$[p(a)][\neg r(Y)]\neg p(Y)$	$ext(9,2,r(Y))$
11	$\square$	$red(10,\neg p(Y))$ et contraction

TAB. 3.4 – *Exemple de démonstration par élimination de modèle*

On remarquera que lors de la production des lemmes à l'étape 8, $[r(Z)]$ n'a pas été promu au rang de lemme, étant donné que le A-littéral situé à sa gauche ($[p(X)]$) est utilisé lors de la réduction. A cet égard, l'objectif des lemmes est d'optimiser la recherche de preuve. Ainsi, le lemme $\neg p(a)$ produit à l'étape 8 pourrait être utilisé avec la chaîne 8 $p(a)$ pour donner directement $\square$.

Dans son principe, l'utilisation des lemmes a pour but d'augmenter l'efficacité de la recherche en identifiant et stockant des solutions pertinentes dont la génération requiert plusieurs étapes d'extension et/ou de réduction.

3.4.3 Hyper-liaison

L'inconvénient majeur du théorème de Herbrand est la multitude des instances qui devront être produites pour trouver une solution. L'idée de base de l'hyper liaison[31] introduite par **Lee** [65] est de générer des instances *proches* d'une instance insatisfiable. Ce principe est similaire aux méthodes de *recherche locale* employées pour la résolution des Problèmes de Satisfaction de Contraintes (voir le chapitre 4).

Au lieu d'utiliser l'énumération de la base de Herbrand, l'hyper-liaison utilise l'unification pour générer des instances (pas nécessairement fondamentales) d'un ensemble de clauses. Ces instances sont ensuite totalement instanciées et on vérifie leur satisfiabilité à l'aide d'un démonstrateur pour le calcul propositionnel.

Définition 3.4.14 - Lien - *Un lien pour un ensemble de clauses S est un couple de littéraux complémentaires unifiables (L,M) appartenant à deux clauses différentes de S. Dans ce cas, les clauses contenant L et M sont dites liées. Une clause est dite totalement liée s'il existe un lien pour chacun de ses littéraux.*

Définition 3.4.15 - Ensemble minimal - *Un ensemble de clauses S est dit* minimal insatisfiable*, s'il n'existe pas de sous-ensemble de S qui soit insatisfiable.*

Par exemple, l'ensemble $S = \{\neg p \vee q, p, \neg q, r \vee t, t \vee p\}$ est insatisfiable mais pas minimal étant donné que le sous-ensemble $\{\neg p \vee q, p, \neg q\}$ est insatisfiable (et minimal).

Théorème 3.4.3 - Théorème d'insatisfiabilité - *Si une instance fondamentale d'un ensemble insatisfiable de clauses S est minimal alors S est totalement lié.*

Preuve voir [24] p.40.

Définition 3.4.16 - Hyper-lien - *Une opération d'hyper-lien consiste à lier totalement une clause $C = L_1 \vee \ldots \vee L_n$, appelée clause noyau (voir figure 3.6) :*

$$\frac{L_1 \vee \ldots \vee L_n \quad M_1 \vee R_1 \ldots M_n \vee R_n}{\sigma(R_1 \vee \ldots \vee R_n)} \; (HL)$$

où σ est un plus grand unificateur des (L_i,M_i),$\forall i, 1 \leq i \leq n$.

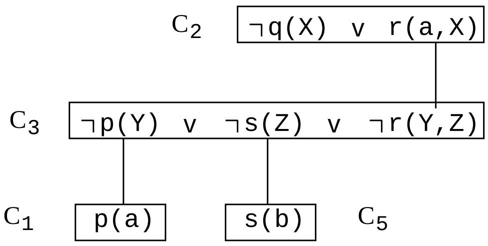

FIG. 3.6 – *Exemple de clause totalement liée*

31. hyper-linking.

Le résultat de l'hyper-lien est une clause non forcément fondamentale. L'objectif de l'hyper-liaison étant de générer et tester des clauses fondamentales, les variables des instances non fondamentales seront substituées par une nouvelle constante (α par exemple).

L'algorithme de la procédure d'hyper-liaison est le suivant :

1. $i = 0, S_0 = S$,
2. si S_i est insatisfiable alors FIN, S est insatisfiable,
3. générer des instances de S_i par hyper-lien. Soit S_i' l'ensemble des instances générées. On pose $S_{i+1} = S_i' \cup S_i$,
4. si $S_{i+1} = S_i$ alors FIN, S est satisfiable,
5. $i = i + 1$, aller à l'étape 2.

Exemple Soit le problème :

$$\begin{aligned} C_1 = &\ p(a) \\ C_2 = &\ \neg q(X) \vee r(a,X) \\ C_3 = &\ \neg p(Y) \vee \neg s(Z) \vee \neg r(Y,Z) \\ C_4 = &\ q(b) \\ C_5 = &\ s(b) \end{aligned}$$

Si la clause C_3 est choisie comme noyau, l'opération d'hyper-lien produit les clauses C_2' et C_3' (voir figure 3.7). L'ensemble des clauses du problème est alors $\{C_1, \ldots, C_5, C_2', C_3'\}$. Les clauses C_2 et C_3 comportent des variables dont les occurrences sont substituées par la constante α. On obtient alors les clauses :

$$\begin{aligned} C_2 = &\ \neg q(\alpha) \vee r(a,\alpha) \\ C_3 = &\ \neg p(\alpha) \vee \neg s(\alpha) \vee \neg r(\alpha,\alpha) \end{aligned}$$

On peut alors déterminer un ensemble de clauses fondamentales insatisfiables pour ce problème, représenté figure 3.7.

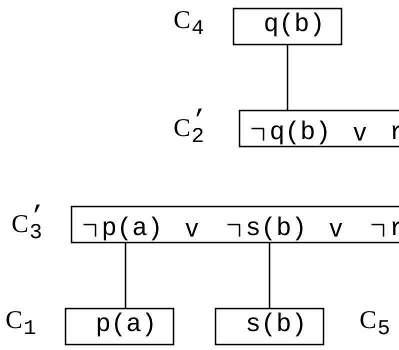

FIG. 3.7 – *Insatisfiabilité avec l'hyper-liaison*

3.4.4 Génération de modèle

La *Génération de modèle*[32] [54] tente de construire un modèle d'un ensemble de clauses S écrites sous forme d'implication (*antécédent* $\rightarrow$ *conséquent*). Ces clauses ont pour particularité d'être *range restricted*, ce qui signifie que tout symbole de variable présent dans le *conséquent* figure également dans l'*antécédent*, assurant ainsi la génération de clauses fondamentales.

32. Model generation.

Si S est satisfiable, un modèle sera exhibé. Par contre, si S est insatisfiable il faudra exploiter toutes les interprétations possibles pour s'assurer qu'aucun modèle n'existe.

Définition 3.4.17 - Notation - *Les clauses de forme générale* $\neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_m$ *sont représentées sous la forme*

$$A_1, \ldots, A_n \rightarrow B_1; \ldots; B_m$$

où les A_i $(0 \leq i \leq n)$ *et* B_j $(0 \leq j \leq m)$ *sont des littéraux.*

Définition 3.4.18 - Clause positive - *Une clause est dite positive si elle ne contient aucun littéral négatif. Elle est alors notée : true* $\rightarrow B_1; \ldots; B_m$.

Définition 3.4.19 - Clause négative - *Une clause est dite négative si elle ne contient aucun littéral positif. Elle est alors notée :* $A_1, \ldots, A_n \rightarrow false$.

Définition 3.4.20 - Clause violée - *Un clause* $A_1, \ldots, A_n \rightarrow B_1; \ldots; B_m$ *est dite violée par une interprétation partielle* $\mathcal{I}$, *s'il existe un plus grand unificateur* σ *tel que :*

$$(\forall i \, (1 \leq i \leq n) \, \sigma(A_i) \in \mathcal{I}) \wedge (\forall j \, (1 \leq j \leq m) \, \sigma(B_j) \notin \mathcal{I})$$

L'algorithme correspondant à la résolution par génération de modèle d'un ensemble de clauses *range restricted* S est le suivant :

1. $\mathcal{I} = \emptyset$
2. trouver une clause violée par $\mathcal{I}$,
   (a) s'il n'en existe aucune, alors FIN, S est satisfiable,
   (b) rejeter $\mathcal{I}$ si la clause est négative,
   (c) sinon pour chaque $\sigma(B_j)$ $(1 \leq j \leq m)$ faire $\mathcal{I} = \mathcal{I} \cup \{\sigma(B_j)\}$ et retourner en 2,
3. si chaque interprétation est rejetée alors FIN, S est insatisfiable.

Exemple La démonstration de l'insatisfiabilité du problème suivant est représentée figure 3.8. Nous avons représenté à chaque noeud de l'arbre, le numéro de la clause violée.

1	$p(X)$	$\rightarrow$	$false$
2	$q(X),s(Y)$	$\rightarrow$	$false$
3	$s(f(X))$	$\rightarrow$	$r(X)$
4	$t(X)$	$\rightarrow$	$r(X)$
5	$r(X)$	$\rightarrow$	$p(X); q(X)$
6	$t(X),q(X)$	$\rightarrow$	$p(f(X)); s(X)$
7	$true$	$\rightarrow$	$s(f(a)); t(b)$

3.4.5 Synthèse

Des méthodes décrites dans ce chapitre, les méthodes que nous qualifions de syntaxico-sémantiques sont les plus efficaces pour prouver l'inconsistance d'un problème. L'explication de cette efficacité réside dans l'utilisation de formes clausales qui retiennent le caractère de satisfiabilité d'un problème et permettent de travailler sur une expression simplifiée de ce dernier.

Du point de vue de l'implantation, les méthodes syntaxiques sont plus contraignantes à mettre en œuvre car elles nécessitent de manipuler des expressions formelles qu'il devient vite difficile à gérer dès lors qu'elles prennent une certaine ampleur. Leur intérêt est qu'elles permettent de raisonner à partir de l'expression logique initiale du problème posé.

Les méthodes sémantiques sont radicalement inefficaces car elles reposent sur la notion d'interprération qui fait appel à des termes totalement instanciés. Le nombre de termes totalement instanciés et par conséquent d'interprétations à étudier est très souvent exponentiel en fonction du nombre de symboles de constantes et fonctions.

Le principe de résolution introduit par J. A. **Robinson** [104] permis d'accroître de manière significative l'efficacité de la résolution des problèmes de la logique des prédicats en autorisant à travailler sur des expressions générales et non plus sur des expressions totalement instanciées.

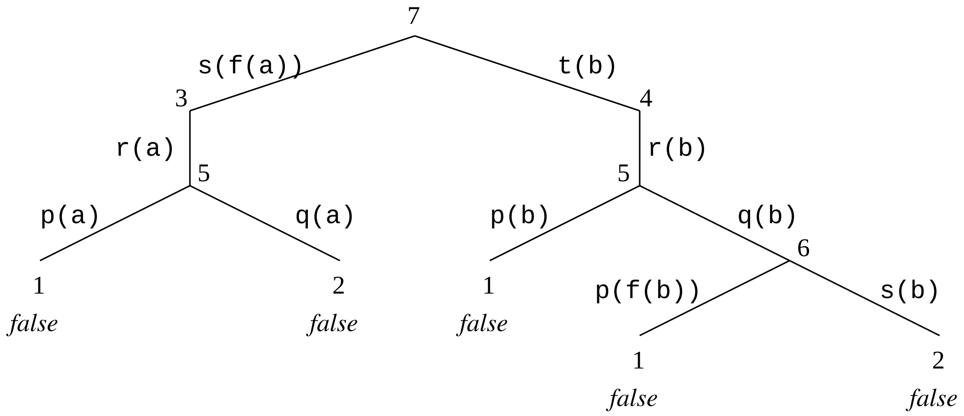

FIG. 3.8 – *Exemple de démonstration avec la génération de modèle*

3.5 Traitement de l'égalité

Le traitement de l'égalité est un problème crucial, notamment si l'on désire s'attaquer à des problèmes touchant au domaine des mathématiques qui font un usage intensif de l'égalité. Lors de la résolution de problèmes avec égalité, les propriétés de la relation d'égalité (symétrie, réflexivité, transitivité et substituabilité) doivent être ajoutées aux axiomes du problème. L'adjonction des schémas d'axiomes de l'égalité augmente la complexité du problème de manière significative. Il faut également noter que les règles d'inférence conçues pour résoudre les problèmes de la Démonstration Automatique sont généralement incapables de prendre en compte le traitement de l'égalité car elle définit en quelque sorte un système de réécriture sur les termes, incompatible avec les règles d'inférence logiques.

Plusieurs méthodes ont été développées avec l'intention d'offrir un traitement adéquat de l'égalité et de réduire le surplus combinatoire introduit par cette relation. Citons en particulier, l'*E-résolution* [85], et la *Paramodulation* [105], la méthode de **Knuth-Bendix** [60], la démodulation [134].

En raison de la difficulté de traitement de la relation d'égalité, des solutions dérivées ont souvent été utilisées, comme l'utilisation de formes prédicatives comme $p(X,Y,Z)$ qui dénote la relation $f(X,Y) = Z$, pour une fonction f. L'associativité peut alors être représentée par le couple de clauses suivantes :

$$p(X,Y,U) \vee \neg p(Y,Z,V) \vee \neg p(U,Z,W) \vee \neg p(X,V,W)$$
$$p(X,Y,U) \vee \neg p(Y,Z,V) \vee \neg p(X,V,W) \vee \neg p(U,Z,W)$$

la commutativité étant représentée par :

$$\neg p(X,Y,Z) \vee p(Y,X,Z)$$

Les formules avec égalité sont construites à partir des termes de la logique du premier ordre. Les axiomes de l'égalité sont des paires *non ordonnées* de termes de la forme $t_1 = t_2$ dont les variables sont universellement quantifiées. Les axiomes de l'égalité sont les suivants :

- **Reflexivité**

$$\frac{}{t = t}\ (Refl)$$

- **Symétrie**

$$\frac{t_2 = t_1}{t_1 = t_2}\ (Sym)$$

- **Transitivité**

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}\ (Trans)$$

- **Congruence**
$$\frac{t_i = s_i,\ \forall i, 1 \leq i \leq n}{f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)}\ (Congr) \quad f \in F$$

- **Substituabilité**
$$\frac{t = s}{\sigma(t) = \sigma(s)}\ (Subst) \quad \text{où } \sigma \text{ est une substitution}$$

Définition 3.5.1 - Notation - *Dans le cadre de cette thèse, nous noterons également* $equal(t_1,t_2)$ *une formule d'égalité.*

Résoudre un problème avec égalité consiste à trouver une procédure de décision pour les théories équationnelles. Illustrons, le problème sur un exemple.

Exemple Considérons un groupe E muni d'une loi $\circ$ qui soit associative, tel que e soit l'élément neutre à droite et où x^{-1} représente l'inverse de x. L'ensemble d'axiomes correspondant à cette théorie est :

$$\begin{array}{lrcl}
(A_1) & x \circ e & = & x \\
(A_2) & x \circ x^{-1} & = & e \\
(A_3) & x^{-1} \circ x & = & e \\
(A_4) & (x \circ y) \circ z & = & x \circ (y \circ z)
\end{array}$$

Une preuve du fait que e soit également l'élément neutre à gauche est :

$$\begin{array}{lll}
e \circ x & = e \circ (x \circ e) & (A_1 gauche) \\
 & = (x \circ x^{-1}) \circ (x \circ e) & (A_2 droite) \\
 & = x \circ (x^{-1} \circ (x \circ e)) & (Trans) \\
 & = x \circ ((x^{-1} \circ x) \circ e) & (A_3 droite) \\
 & = x \circ (e \circ e)) & (A_1) gauche)\ x = e \\
 & = x \circ e &
\end{array}$$

Automatiser le traitement de l'égalité consiste à déterminer quel axiome appliquer à chaque étape de la preuve, et où l'appliquer. Une égalité n'étant pas orientée, on réécrit, comme le montre l'exemple précédent, tantôt en fonction de la partie gauche de l'égalité et tantôt en fonction de la partie droite. On peut éventuellement entrer dans une boucle infinie, la partie droite annulant l'effet de la partie gauche et inversement. L'utilisation d'un système de réécriture permet d'orienter les égalités et s'avère primordial car on supprime ainsi le tâtonnement, c'est-à-dire l'application inutile de certaines règles à certaines étapes de la preuve. Par exemple, un système de réécriture pour la théorie des groupes est donné dans [60].

3.5.1 Paramodulation

La règle de paramodulation [105] combinée à la règle de résolution forme un système complet pour la réfutation. La règle de paramodulation s'énonce :

$$\frac{C_1 = equal(t_1,t_2) \vee C_1' \qquad C_2 = L_2[t] \vee C_2'}{\sigma(L_2[t_2] \vee C_1' \vee C_2')}\ (Para)$$

où σ est un plus grand unificateur de $(t_1,L_2[t])$ et la notation $L_2[t]$ signifie que le terme t est un sous-terme du littéral L_2.

De nombreux raffinements ont été proposés pour la paramodulation, comme la règle de *superposition* qui interdit tout remplacement dans le plus petit membre d'une équation, ou les règles de *simplification clausale* [108] (inspirées de [60]) éliminant les clauses redondantes.

Néanmoins, certains problèmes restent incontournables, notamment pour les fonctions commutatives ($equal(f(X,Y),f(Y,X))$ et associatives telles que $equal(f(f(X,Y),Z),f(X,f(Y,Z)))$ (voir [123], p. 12). En effet si f est une fonction associative et commutative, les représentations $f(a,f(b,c))$, $f(a,f(c,b))$, $f(b,f(a,c))$, $f(f(a,c),b),\ldots$ sont toutes équivalentes .

Exemple En guise d'exemple, appliquons la paramodulation entre $C_1 = equal(g(X),f(X,b)) \vee p(f(X,Y))$ et $C_2 = q(f(g(a),b))$:

$$\frac{equal(g(X),f(X,b)) \vee p(f(X,Y)) \qquad q(f(g(a),b))}{p(f(a,Y)) \vee q(f(f(a,b),b))}\ (Para)$$

Si $C_1 = equal(f(X,b),g(X)) \vee p(f(X,Y))$, $C_2 = q(f(g(a),b))$, on a :

$$\frac{equal(g(X),f(X,b)) \vee p(f(X,Y)) \qquad q(f(g(a),b))}{p(f(g(a),Y)) \vee q(g(g(a)))}\ (Para)$$

3.5.2 Démodulation

À un degré moindre, la *démodulation* [134] est un procédé de réécriture canonique des clauses en fonction de clauses unitaires égalitaires, appelés *démodulateurs*.

Elle s'organise suivant une orientation des clauses égalitaires de manière à ce que la partie droite de la clause d'égalité soit "plus courte" (en nombre de symboles de fonctions et de variables) que sa partie gauche. Ainsi, un démodulateur de la forme $equal(h(X),f(f(X)))$ sera réorienté en $equal(f(f(X)),h(X))$. Certains termes sont cependant difficiles à comparer, comme par exemple $h(f(X),Y)$ et $g(X,f(Y))$. On a donc recours à des techniques aptes à ordonner et classer les termes, comme la technique de LRPO [33] [34] qui jouit de propriétés théoriques intéressantes (comme la terminaison). On peut également utiliser l'algorithme de **Knuth**, **Bendix** [60] qui tente de transformer un ensemble de démodulateurs en un système de réécriture canonique qui termine. La terminaison est essentielle car elle garantit que l'application récursive des démodulateurs sur un littéral terminera.

Exemple Supposons que la liste des démodulateurs soit :

$$\begin{array}{ll} (d_1) & equal(g(b,a),a) \\ (d_2) & equal(h(a),b) \\ (d_3) & equal(f(f(X)),h(X)) \end{array}$$

alors, l'application de la démodulation sur $p(f(f(a)),g(h(X),a))$ donne successivement :

$$\begin{array}{ll} p(h(a),g(h(a),a)) & \textit{application de } (d_3) \\ p(h(a),g(b,a)) & \textit{application de } (d_2) \\ p(h(a),a) & \textit{application de } (d_1) \end{array}$$

La démodulation demande que les variables des littéraux à démoduler restent inchangées. Ainsi pour

$$\begin{array}{l} equal(father(pete),steve) \\ older(father(X),X) \end{array}$$

la démodulation ne permet pas d'obtenir la clause suivante (celle-ci peut être cependant obtenue par paramodulation) :

$$older(steve,pete)$$

3.6 Démonstrateurs automatiques

Dans cette section, nous décrivons quatre logiciels pour la Démonstration Automatique. Une grande variété de démonstrateurs pour la logique des prédicats du premier ordre existe à l'heure actuelle. Notre choix s'est porté sur Otter, SETHEO, Satchmo et CLIN-S car ils implantent les procédures de preuve présentées précédemment. En outre, ces logiciels ont pris part à la compétition organisée en marge de la conférence internationale *CADE-13* qui s'est déroulée le 1^{er} Août 1996 à l'université de Rutgers, New Jersey, USA. Les résultats dont nous disposons nous permettrons d'établir un comparatif entre ces démonstrateurs automatique.

33. lexicographic recursive path ordering.

3.6.1 Otter

Otter[34] [81, 80] est un logiciel de démonstration automatique pour la logique des prédicats du premier ordre avec égalité.

Otter fait partie de la quatrième génération de démonstrateurs automatiques développés par la division Mathématiques et Informatique du *Argonne National Laboratory*, organisme de recherche américain. Les recherches concernant le Raisonnement Automatique ont débuté à Argonne (IL, USA) en 1962. Otter, qui fut opérationnel en Décembre 1987 est l'aboutissement de 25 ans d'expérience dans le domaine de la Démonstration Automatique.

La deuxième génération de démonstrateurs automatiques, initiée en 1970 par Ross **Overbeek**, basée sur l'architecture NIUTP[35] [79] a donné naissance à AURA[36] [113]. Le couple NIUTP/AURA a permis de répondre à un certain nombre de questions restées ouvertes dans le domaine du calcul d'équivalence [130], les algèbres booléennes ternaires [127] ou les semi-groupes [128].

La troisième génération, formée de LMA[37] [73] et ITP[38] a constitué une base de techniques effectives pour la réalisation d'Otter qui retient la plupart des fonctionnalités de ces logiciels, plus particulièrement en ce qui concerne les règles d'inférence et les méthodes de recherche.

Otter s'appuie sur un ensemble de règles d'inférences s'articulant autour du *principe de résolution*: résolution binaire, hyperrésolution, hyperrésolution négative, *unit-resulting* résolution et *linked unit-resulting* résolution. Le traitement de l'égalité est réalisé par l'emploi de la paramodulation binaire ainsi que des variantes de la démodulation.

Tout comme ses prédécesseurs (AURA, LMA, ITP), Otter utilise un algorithme, appelé *given-clause algorithm* (donné figure 3.9) qui peut être considéré comme une implantation simplifiée de la stratégie de l'ensemble support. Pour cela, Otter maintient quatre listes de clauses :

1. *usable* : cette liste est formée des clauses qui sont choisies lors de l'application des règles d'inférence,
2. *sos* : les clauses de la liste sos (Set of Support) sont utilisées pour poursuivre la recherche,
3. *passive* : cette liste définie dans la spécification
4. *demodulators* : ce sont des clauses utilisées comme règles de réécriture sur les nouvelles clauses inférées.

```
tantque sos n'est pas vide et qu'aucune réfutation n'a été trouvée
    Choisir une clause c ∈ sos
    Déplacer c de sos vers usable
    Appliquer sur c les règles d'inférence en cours
        chaque nouvelle clause est insérée dans sos et doit
        obligatoirement avoir en tant que parents : la clause c et d'autres clauses de usable.

fin
```

FIG. 3.9 – *given-clause algorithm d'Otter*

Les nouvelles clauses obtenues par application des règles d'inférence subissent un ensemble de tests de rétention qui ont pour objet d'éliminer les clauses redondantes ou les clauses qui ne répondent pas à des critères de rétention fournis par l'utilisateur :

- abandon de la clause si son poids est supérieur à une valeur donnée (*weighting*)ou si le nombre de variables ou de littéraux est trop important,
- abandon de la clause si elle est subsumée par une clause de *sos*,
- recherche de conflits avec l'ensemble des clauses de *sos* et *usable*,

34. Organized Techniques for Theorem-proving and Effective Research
35. Northern Illinois University Theorem Prover
36. AUtomated Reasoning Assistant
37. Logic Machine Architecture
38. Interactive Theorem Prover

Dans le cas ou la spécification utilise une théorie égalitaire, un certain nombre d'autres traitements viennent s'ajouter à ceux que nous venons de citer :

- orientation des clauses égalitaires,
- détermination de la validité de la clause en tant que démodulateur et ajout à *demodulators* dans ce cas,
- démodulation en fonction d'un membre de *demodulators*,
- pour les clauses égalitaires ajout à la liste *demodulators* et réécriture des clauses de *sos* et *usable* en fonction du nouveau démodulateur,

3.6.2 Satchmo

Satchmo [76, 48] est fondé sur la génération de modèle et a été développé, à l'origine, pour tester la satisfiabilité de contraintes dans les bases de données déductives. La première version de Satchmo fut implanté en Prolog, tirant ainsi profit des caractéristiques de Prolog dans les domaines des structures de termes, du mécanisme d'unification et du retour arrière. Le code de Satchmo dans sa première version le rend très attractif car il est composé d'une dizaine de règles Prolog.

Partant de la constatation qu'un démonstrateur automatique peut être considéré comme un interpréteur d'un programme, des techniques de compilation ont été utilisées afin de gagner en efficacité et une variante, appelée *Compiling Satchmo*, a vu le jour.

Une deuxième variante, *Functional Satchmo* a été implantée dans un langage fonctionnel proche de *Scheme* afin d'intégrer plus facilement des modifications et des optimisations qu'en Prolog.

Ces deux variantes utilisent les *positive unit hyperresolution* (PUHR) tableaux [16].

3.6.3 Setheo

L'élimination de modèle a connu un regain d'intérêt avec les travaux de **Stickel** qui permirent de la réhabiliter après être tombée aux oubliettes pendant quelques années. Le *Prolog Technology Theorem Prover* (PTTP) [115, 116] montra que l'élimination de modèle était une alternative intéressante à l'hyperrésolution et aux autres règles d'inférence basées sur le principe de résolution.

SETHEO [5, 48] est basé sur la méthode des *connection tableaux* [67] qui est en quelque sorte une méta-théorie de l'élimination de modèle dans le cadre des tableaux sémantiques. SETHEO est implanté comme une extension de la *Machine Abstraite de Warren* [126].

Parmi les traits caractéristiques de SETHEO, on distingue : l'utilisation des mécanismes de cache, de lemmes et l'emploi d'une stratégie de recherche par approfondissements successifs, en l'occurrence la *consecutively bounded depth-first iterative deepening search* [117], qui prend en compte le nombre d'inférences produites à l'étape précédente lors de la modification de la borne de profondeur.

Un mécanisme de cache complet, qualifié de *local failure caching* [52] accélère la recherche en évitant de recalculer des déductions obtenues auparavant par les règles d'inférence. Le *caching* a pour but de remplacer la recherche en stockant les résultats précédemment produits et qui n'ont plus à être recalculés, ce qui a pour effet de diminuer le temps de calcul. L'emploi des lemmes (*lemmaizing*) a été expliqué en section 3.4.2. L'idée est de garder des solutions intermédiaires (ou lemmes) qui seront utilisées afin d'éviter des sous-déductions répétées.

SETHEO utilise en outre une méthode dite de *pertinence*[39] [99] sensée choisir les clauses qui feront effectivement partie d'une preuve. Une clause c est dite pertinente pour un ensemble de clauses S, s'il existe un sous-ensemble satisfiable $S' \subset S$ tel que $S' \cup \{c\}$ est insatisfiable. La pertinence se traduit, par exemple, par le choix d'une clause négative au départ de la preuve, sachant que dans la plupart des cas on tente de montrer que $H \cup \neg F$ est insatisfiable.

L'une des faiblesses de l'élimination de modèle est le traitement de l'égalité. En effet, la paramodulation, qui est très efficace, n'est pas complète pour les ensembles orientés de clauses. Une variante de SETHEO, appelée E-SETHEO permet de traiter les problèmes avec clauses égalitaires. E-SETHEO est basé sur la *lazy basic paramodulation* [44].

39. Relevance en anglais.

3.6.4 CLIN-S

Lee a développé le démonstrateur automatique CLIN [65] basé sur l'hyper-liaison dont l'algorithme est donné figure 3.10. **Chu** a ensuite développé CLIN-S [25] qui combine l'hyper-liaison avec une énumération de la base de Herbrand [24]. CLIN-S se veut être un démonstrateur guidé par la sémantique. Il se fondé sur l'idée que la preuve de théorèmes devrait reposer sur le sens (sémantique) de concepts pertinents pour la recherche et non sur la forme de représentation (syntaxe) de ces concepts.

CLIN-S utilise l'hyper-liaison sémantique[40] qui est un procédé de recherche de modèle d'un ensemble de clauses et qui peut être considéré comme une recherche dans un arbre sémantique clos. La recherche est effectuée de manière incrémentale de façon à éviter les modèles redondants. En particulier, les branches closes de l'arbre ne seront plus revisitées.

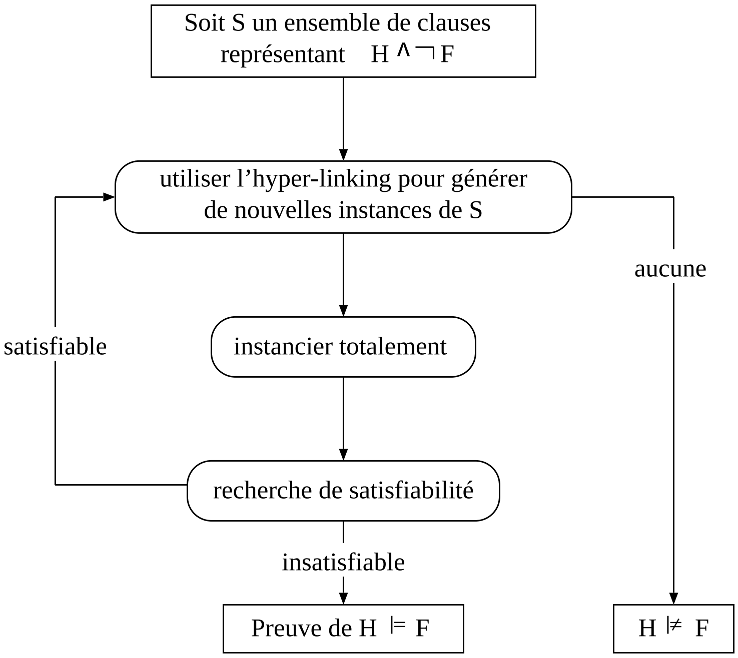

FIG. 3.10 – *Algorithme de l'Hyper-linking.*

CLIN-S a également recours à la *rough resolution* et à l' *unit-resulting resolution*. La *rough resolution*, que l'on pourrait traduire par *résolution grossière*, est une restriction du principe de résolution qui se fonde sur la génération de "petites" résolvantes en résolvant les littéraux les plus grands en nombre de symboles fonctionnels dans les clauses parentes. On évite ainsi la génération sur la base de Herbrand de termes complexes. La résolution unitaire de type *unit-resulting* est, quant à elle, utilisée pour résoudre les ensembles de clauses de Horn. Elle sert en particulier à détecter une contradiction qui résulterait de la construction d'un modèle.

3.7 Conclusion

Le moyen le plus empirique mais également le plus pratique pour comparer des démonstrateurs est de les soumettre à une batterie de tests et de mesurer leurs performances sur chacun de ces problèmes.

Le moyen le plus approprié pour tester l'efficacité d'une procédure de preuve est d'implanter celle-ci au travers d'un logiciel. Seule une implantation logicielle permet de trancher de manière décisive quant à l'efficacité de la procédure, même si résultats théoriques escomptés sembles prometteurs. Deux logiciels implantant la même procédure de preuve sont susceptibles d'amener des résultats totalement différents. L'un peut se révéler plus efficace que l'autre. Un grand nombre de facteurs sont capables d'influencer

40. semantic hyper-linking.

les performances d'un logiciel : en premier lieu on trouve le langage d'implantation, puis viennent les structures de données employées et les optimisations consenties. L'implantation logicielle représente la plus haute expression de l'ingéniosité et de l'habileté du programmeur à concevoir des structures de données et des algorithmes efficaces.

3.7.1 La compétition CADE-13

La compétition qui eut lieu le 1er Août 1996 à l'université de Rutgers (New Jersey, USA), en marge de la conférence internationale CADE-13 a vu s'affronter 18 systèmes de démonstration automatique pour la logique des prédicats du premier ordre. Les motivations de cette compétition consistaient à contribuer à l'évaluation des systèmes de démonstration automatique, stimuler le développement et la recherche autour de ces systèmes et exposer les systèmes actuels aux chercheurs faisant partie ou non du domaine de la Démonstration Automatique.

Concevoir et gérer une telle compétition est, selon les propos de ces organisateurs, une difficile et contestable gageure, mais se révèle également une expérience enrichissante et stimulante. L'hétérogénéité des systèmes de démonstration automatique et l'absence de critères de comparaison véritables (soulignée par **Overbeek** dans [91]) soulève d'emblée le problème de la définition des modalités des procédures de tests et de la recherche de critères non équivoques, capables de déterminer les vainqueurs.

L'ensemble des modalités de la compétition CADE-13 est décrit dans [118]. Les systèmes ayant pris part à la compétition devaient répondre à un certains nombres de caractéristiques, parmi lesquelles :

- aucune interaction humaine n'est possible, le système est livré à lui même,
- le système doit être adéquat, mais pas forcément complet
- le système fournit toujours les mêmes résultats en sortie pour les mêmes données en entrée.

La totalité des résultats obtenus lors de la compétition CADE-13 est largement discuté dans [119]

3.7.2 Notion d'efficacité

Nous définissons enfin un critère de comparaison empirique de l'efficacité des démonstrateurs automatiques que nous utiliserons par la suite pour comparer nos résultats.

Définition 3.7.1 - Efficacité d'un démonstrateur - *Soit une batterie de tests composée de M problèmes. Si un démonstrateur automatique résout, en T secondes, N problèmes parmi les M qui lui sont proposés, alors son efficacité est donnée par la formule :*

$$e = \frac{N^2}{T}$$

On aurait pu choisir comme formule de l'efficacité $e = N/T$, mais alors un démonstrateur (da_1) résolvant 10 problèmes en $100s$ aurait obtenu la même efficacité qu'un autre démonstrateur (da_2) venant à bout de 100 problèmes en $1000s$. Intuitivement, un démonstrateur capable de résoudre un panel plus vaste de problèmes est sensé être plus performant. Nous avons donc donné la priorité au nombre de problèmes résolus en élevant N au carré. (da_1) et (da_2) ont alors pour efficacités respectives 1 et 10. La formule choisie pour l'efficacité nous paraît représenter un compromis équilibré entre le nombre de problèmes résolus et temps de résolution.

Nous avons fait figurer table 3.5 l'efficacité de certains démonstrateurs de théorèmes ayant pris part à la compétition de la conférence $CADE'13$. Ces démonstrateurs sont décrits dans [129] En fonction du critère d'efficacité choisi, le système Gandalf se révèle le plus efficace bien que résolvant moins de problèmes qu'Otter ou SETHEO. Cependant le temps de résolution de ces problèmes est nettement moindre que celui de SETHEO par exemple.

Démonstrateur	Pb Résolus	Temps	Efficacité
Gandalf	25	370	1.69
Otter	32	637	1.61
Violet	30	694	1.30
SPASS	32	844	1.21
Otter 3.0.4z	28	714	1.10
SETHEO	31	945	1.02
E-SETHEO	36	1455	0.89
Comp-Satchmo	6	53	0.68
Linus	25	945	0.66
Func-Satchmo	4	41	0.39
RRTP	21	2038	0.22
CLIN-S	9	1176	0.07

TAB. 3.5 – *Démonstrateurs classés suivant leur efficacité*

Chapitre 4

Problèmes de Satisfaction de Contraintes

L'objectif de ce chapitre est d'introduire de manière très générale les notions associées aux Problèmes de Satisfaction de Contraintes, tout en insistant sur les similitudes qui peuvent exister avec la Démonstration Automatique dans le cadre des méthodes de résolution de ces problèmes, sans toutefois en occulter les spécificités.

4.1 Introduction

Un grand nombre de problèmes de l'Intelligence Artificielle, ainsi que d'autres domaines de l'informatique, sont modélisables sous forme de Problèmes de Satisfaction de Contraintes[41] (CSP). Parmi ces problèmes on distingue les problèmes liés à la vision artificielle [75, 84], à la conception de circuit [33], à la planification et l'ordonnancement [35] au diagnostic de pannes [47].

La **Programmation Par Contraintes** fournit un cadre de représentation et de résolution des problèmes axé sur des contraintes exprimées entre des variables évoluant sur un domaine de valeurs. Si ce cadre peut sembler quelque peu *dépouillé*, il permet néanmoins d'exprimer de manière relativement simple un grand nombre de problèmes. L'intérêt d'une représentation contrainte est double. D'une part, la description des problèmes est déclarative, on doit en effet décrire les conditions que doivent remplir les solutions du problème et non pas le moyen de calculer les solutions. Tout comme en Programmation en Logique, on spécifie le quoi sans se soucier du comment. Autrement dit, l'expression du problème est neutre par rapport au système de résolution de contraintes. On peut de plus couplé l'expression des contraintes à d'autres langages, comme par exemple en **Programmation Logique avec Contraintes** pour obtenir une formulation de haut niveau d'abstraction.

Plusieurs systèmes de Programmation Par Contraintes on été développés. Parmi lesquels *Prolog III* [29], *Chip* [36, 121], *CLP(R)* [57], *Charme* [89], *Ilog Solver* [97] et dans une moindre mesure *Alice* développé par J.-L. **Laurière** [64].. Les domaines sur lesquels évoluent les variables sont assez variés, on trouve, entre autres, les domaines finis booléens et entiers, les rationnels, les réels, mais également les ensembles [66, 2].

4.2 Syntaxe

Définition 4.2.1 - Problème de satisfaction de contraintes - *Un problème de satisfaction de contraintes $\mathcal{P} = \langle \mathcal{V},\mathcal{D},\mathcal{C} \rangle$ est défini par :*

- *un ensemble $\mathcal{V} = \{x_1, \ldots ,x_n\}$ de n variables,*

41. Constraint Satisfaction Problem.

- *un ensemble* $\mathcal{D} = \{D_1, \ldots ,D_n\}$ *de* n *domaines finis, tels que* x_i *prend ses valeurs dans* D_i, $\forall i$ $(1 \leq i \leq n)$,
- *un ensemble* $\mathcal{C} = \{C_1, \ldots ,C_m\}$ *de* m *contraintes ; chaque contrainte* C_i *est définie par un couple* (V_i,R_i) *tel que :*
  - V_i *est un sous-ensemble de* $\mathcal{V}$*, la cardinalité de l'ensemble* V_i *étant appelée arité de la contrainte,*
  - R_i *est un sous-ensemble du produit cartésien des domaines des variables de l'ensemble* V_i.

On utilise également une représentation graphique des CSP sous forme d'hypergraphe, permettant ainsi d'expliciter la structure du problème.

Définition 4.2.2 - Hypergraphe - *Un hypergraphe* $G = \langle P,A \rangle$ *est composé*
- *d'un ensemble* $P = \{p_1, \ldots ,p_n\}$ *fini de sommets,*
- *d'un ensemble* $A = \{a_1, \ldots ,a_m\}$ *de parties de* P*, appelées hyperarêtes de l'hypergraphe, telles que* $\cup_{i=1}^{i=m} a_i = P$.

De manière générale, P correspond à l'ensemble des variables du CSP et A aux contraintes du problèmes. Ce type de représentation est particulièrement bien adapté aux CSP binaires. Pour les CSP n-aires $(n > 2)$, on préfére utiliser un graphe biparti associé à l'hypergraphe.

Définition 4.2.3 - Graphe biparti - *Soit* $G = \langle P,A \rangle$ *un hypergraphe. Le graphe biparti* $G_B = \langle P_B,A_B \rangle$ *associé à* G *est défini de la manière suivante :*
- $P_B = P \cup A$,
- $A_B = \{(p_i,a_j)$ *tels que* $p_i \in P,a_j \in A\}$.

Exemple Nous avons dessiné figure 4.1 les représentations du problème (P_1) suivant sous forme de graphe biparti et d'hypergraphe.

$$(P_1)\left\{\begin{array}{l} \mathcal{V} = \{x_1,x_2,x_3,x_4,x_5\} \\ \mathcal{D} = \{D_1 = D_2 = D_3 = D_4 = D_5 = \{1..10\} \\ \mathcal{C} = \left\{\begin{array}{ll} (C_1) & x_1 + x_2 < x_3 \\ (C_2) & x_2 - x_4 < x_5 \\ (C_3) & x_1 > x_3 \end{array}\right\} \end{array}\right.$$

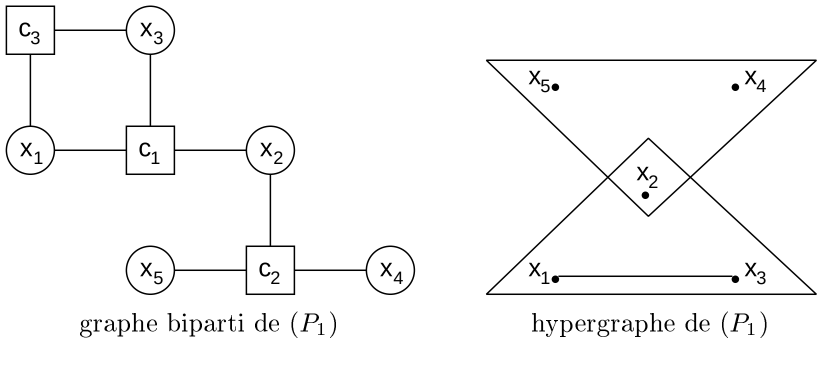

FIG. 4.1 – *Exemples d'hypergraphe et de graphe biparti.*

Dans la suite de ce chapitre nous axons notre exposé sur un ensemble de propriétés restreintes aux CSP binaires afin de simplifier les notions et notations employées. Tout CSP n-aire $(n > 2)$ est susceptible d'être exprimaé sous forme de CSP binaire équivalent [106] dont les hyperarêtes sont couples de variables de la forme (x_i,x_j).

Définition 4.2.4 - CSP binaire - *Un CSP binaire est un CSP dont toutes les contraintes sont d'arité égale à 2.*

Les hyperarêtes de l'hypergraphe associé à un CSP binaire seront notées $(x_i \rightleftharpoons x_j)$ par la suite. L'existence d'une contrainte binaire entre deux variables x_i et x_j pour le CSP implique l'existence d'une hyperarête au niveau de l'hypergraphe associé. Enfin précisons, en prévision de ce que nous verrons dans la suite du chapitre, que la notion d'arête d'un graphe est non orientée. Elle représente un chemin entre deux sommets du graphe. Il est parfois nécessaire d'utiliser la notion d'arc qui est unidirectionnelle. Ainsi, une arête $(x_i \rightleftharpoons x_j)$ est donc composée de deux arcs $(x_i \rightharpoonup x_j)$ et $(x_j \rightharpoonup x_i)$.

4.3 Sémantique

Définition 4.3.1 - Instanciation - *On appelle instanciation $\mathcal{I}$ d'un sous ensemble de variables $V = \{x_1,\ldots,x_k\} \subset \mathcal{V}$, une application qui à chaque variable x_i,$(1 \leq i \leq k)$ associe une valeur $d_i \in D_i$. On notera $\mathcal{I}(V)$ l'assignation de chaque élément de V par la valeur qui lui est associée dans $\mathcal{I}$.*

Définition 4.3.2 - Configuration - *Une configuration est une instanciation qui porte sur l'ensemble des variables du problème.*

Définition 4.3.3 - Satisfaction de contrainte - *Une contrainte $C_i = (V_i,R_i)$ est dite satisfaite s'il existe une instanciation $\mathcal{I}$ du sous-ensemble de variables V_i sur lesquelles elle s'exprime, telle que $\mathcal{I} \in R_i$. On note alors $\mathcal{I} \models C_i$.*

Définition 4.3.4 - Instanciation consistante - *Une instanciation $\mathcal{I}$ d'un sous-ensemble de variables $V \subset \mathcal{V}$ est dite consistante si et seulement si pour chaque contrainte $C_i = (V_i,R_i)$ telle que $V_i \subset V$, on a $\mathcal{I} \models C_i$.*

Définition 4.3.5 - Solution d'un CSP - *Une configuration consistante $\mathcal{I}$ de $\mathcal{V}$ est appelée solution de $\mathcal{P}$. Dans ce cas, on note $\mathcal{I} \models \mathcal{P}$. L'ensemble des solutions de $\mathcal{P}$ est quant à lui noté $S_{\mathcal{P}}$.*

Définition 4.3.6 - Consistance d'un CSP - *Un CSP est dit consistant ou satisfiable s'il possède une solution.*

Définition 4.3.7 - Instanciation globalement consistante - *Une instanciation $\mathcal{I}$ d'un sous-ensemble de variables $V \subset \mathcal{V}$ est dite globalement consistante si et seulement si il existe une instanciation $\mathcal{J}$ de $\mathcal{V} \setminus V$ qui soit une solution de $\mathcal{P}$.*

Une instanciation globalement consistante est donc une sous-solution, ou solution partielle du problème. Si une instanciation est non globalement consistante, alors elle ne fera jamais partie d'une solution.

Définition 4.3.8 - CSP équivalents - *Deux CSP $\mathcal{P}$ et $\mathcal{Q}$ sont dits équivalents ($\mathcal{P} \equiv \mathcal{Q}$) si et seulement si $S_{\mathcal{P}} = S_{\mathcal{Q}}$.*

Résoudre un CSP consiste à exhiber une configuration globalement consistante des variables du problème. Mais il existe d'autres questions auxquelles on peut vouloir répondre à propos d'un CSP :

- déterminer une solution (recherche de consistance ou de satisfiabilité),
- déterminer toutes les solutions,
- rechercher une solution optimale pour une fonction objectif donnée (optimisation),
- rechercher une configuration satisfaisant le plus de contraintes possible (problème sur-contraint) [42],
- estimer le nombre de solutions [4, 38].

Le problème de la recherche d'une solution est NP-complet.

4.4 Techniques de résolution

Tout comme en Démonstration Automatique, nous sommes en mesure d'exhiber plusieurs critères de différenciation permettant d'établir une classification des méthodes de résolution des CSP. On distingue notamment :

- les méthodes *systématiques* ou *non systématiques*. Le principe des méthodes systématiques consiste à parcourir l'arbre de recherche associé au problème en visitant chaque branche une seule fois. Les CSP étant composés de domaines finis, l'arbre de recherche est donc de profondeur finie. Ces méthodes sont complètes : on trouvera une solution s'il en existe une. Par contre, si le problème est insatisfiable, l'arbre de recherche sera couvert totalement pour le démontrer. Par opposition, les méthodes non systématiques n'examinent pas toutes les branches de l'arbre, elles sont donc incomplètes.
- les méthodes *constructives* ou *locales*. Une méthode constructive instancie successivement les variables du problème de manière à toujours travailler avec une solution partielle. Les méthodes locales, au contraire, manipulent une configuration non consistante qu'elles tentent de rendre consistante par des modifications successives.
- les méthodes *déterministes* ou *stochastiques*. Les méthodes déterministes fournissent toujours la même trace d'exécution pour les mêmes données en entrée, alors que pour les méthodes stochastiques la trace dépend de choix associés à des probabilités.

Le critère que nous avons choisi pour présenter les méthodes de résolution des CSP est le critère systématique/non systématique qui semble rendre compte de la diversité des méthodes avec plus de clareté .

4.5 Méthodes systématiques

4.5.1 Générer / Tester

La méthode *générer/tester*[42] (GT) est la plus naturelle et la plus simple à mettre en oeuvre dans le cadre des CSP, elle est donc, parmi les méthodes que nous présenterons, la moins efficace. Si l'on désire dresser un parallèle avec la Démonstration Automatique, la méthode générer/tester s'apparente aux méthodes basées sur le théorème de Herbrand. Elle consiste en une énumération de toutes les configurations possibles suivie d'une vérification de consistance de chaque instance (voir algorithme figure 4.2).

```
procedure GT
debut
    𝓘 = { ensemble des configurations du problème }
    tantque 𝓘 ≠ ∅ faire
        choisir I ∈ 𝓘
        𝓘 = 𝓘 \ {I}
        si I est consistante alors Solution fsi
    ftq
fin
```

FIG. 4.2 – *Algorithme GT.*

Si l'on considère, de manière très générale, qu'un CSP est composé de n variables, de m contraintes, et que chaque domaine comporte d valeurs, le nombre de configurations à tester pour démontrer l'inconsistance d'un problème suivant GT est au maximum de d^n. En conséquence, on peut être amené à évaluer un maximum de $m \times d^n$ contraintes.

42. *generate and test* en anglais

Exemple Considérons le problème suivant :

$$(P_2)\left\{\begin{array}{l} \mathcal{V} = \{x_1,x_2,x_3\} \\ \mathcal{D} = \left\{\begin{array}{l} D_1 = \{2,3,4\}, \\ D_2 = \{1,2,3\}, \\ D_3 = \{2,3\} \end{array}\right\} \\ \mathcal{C} = \left\{\begin{array}{ll} (C_1) & x_1 > x_2 \\ (C_2) & x_2 < x_3 \\ (C_3) & x_1 < x_3 \end{array}\right\} \end{array}\right.$$

Nous avons représenté figure 4.3, l'ensemble des configurations du problème (P_2) sous forme d'un arbre de recherche. Ce type de représentation arborescente s'adapte parfaitement au cadre de résolution des méthodes systématiques car il établit clairement le cheminenment menant à la découverte d'une solution.

Lorsqu'un problème de taille conséquente ne possède que quelques solutions (voire aucune), le temps de résolution peut devenir prohibitif. Il est donc primordial de concevoir des méthodes plus efficaces.

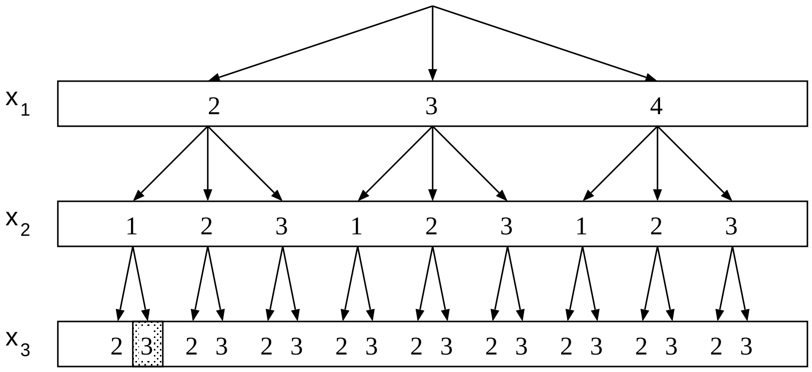

FIG. 4.3 – *Résolution de (P_2) suivant GT*

4.5.2 Le retour arrière

Les méthodes systématiques se fondent sur le principe du retour arrière[43] (BT) [11]. Ce principe consiste à construire une solution en instanciant les variables du problème suivant un ordre préalablement établi. L'instanciation se poursuit tant qu'elle est consistante. Si l'instanciation se révèle inconsistante, on remet en cause la valeur de la dernière variables instanciée. Si toutes les valeurs du domaine d'une variable conduisent à des instanciations inconsistantes, on remet en cause l'instanciation de la variable précédemment instanciée.

Nous donnons figure 4.4, l'algorithme correspondant au retour arrière pour lequel x_i est la variable courante à instancier, X_N est l'ensemble des variables non instanciées et X_I est l'ensemble des variables instanciées.

La figure 4.5 représente l'élagage réalisé par le retour arrière. On voit clairement, en comparaison de la méthode GT, que 6 configurations ont été éliminées. L'arbre de recherche ne contient plus que 24 noeuds et 12 configurations, alors qu'il comporte 30 noeuds et 18 configurations sur la figure 4.3.

Certes, le retour arrière gagne en efficacité par rapport à la méthode générer/tester, mais il souffre de certains défauts connus sous l'appellation de *Thrashing*[44] : c'est-à-dire qu'il échoue plusieurs fois pour les mêmes raisons dans l'établissement d'une configuration consistante. De manière plus précise, l'algorithme BT effectue un parcours *aveugle* de l'espace de recherche sans prendre en compte des informations *évidentes* sur l'inconsistance globale d'instanciations partielles et redécouvre, de fait, de manière répétée, les mêmes inconsistances locales.

43. *backtrack* en anglais, parfois qualifié de *test and generate*.
44. que l'on pourraît traduire par amnésie.

procedure BT(X_I,X_N,$\mathcal{I}$)
entree variables instanciées X_I et non instanciées X_N, instanciation courante $\mathcal{I}$
debut
  si $X_N = \emptyset$ alors Solution fsi
  choisir $x_i \in X_N$ la prochaine variable à instancier
  pour chaque valeur $d_i \in D_i$ faire
    si $\mathcal{I} \cup \{x_i = d_i\}$ est consistante alors
      BT($X_I \cup \{x_i\}$,$X_N \setminus \{x_i\}$,$\mathcal{I} \cup \{x_i = d_i\}$)
    fsi
  fpour
fin

FIG. 4.4 – *Algorithme du backtrack.*

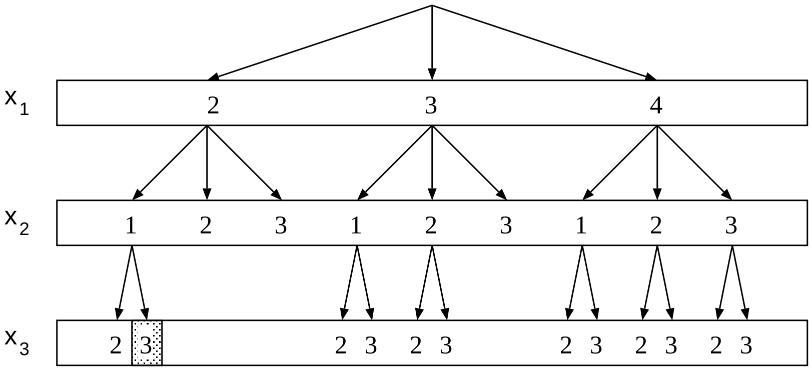

FIG. 4.5 – *Résolution avec BT*

De nombreuses recherches ont été effectuées afin d'améliorer la prise en compte des causes d'inconsistances pour le retour arrière. On distingue :

- *les techniques de simplification utilisées avant la recherche :* ces techniques se fixent pour objectif d'obtenir un CSP équivalent au CSP initial, mais de forme plus simple. On distingue à cet échelon, les techniques dites de **filtrage**, liées à la notion de **consistance locale**, et les techniques de **décomposition** (cf. [1].
- *les techniques utilisées durant la recherche :* on distingue les techniques de **look ahead** utilisant le filtrage sur une instanciation partielle, et les techniques **look back** qui en cas d'échec tentent de revenir à la cause réelle de l'échec et non pas à la dernière variable instanciée.
- *les techniques utilisées pour guider la recherche :* ce sont des techniques heuristiques fondées sur l'ordonnancement des variables et des valeurs.

4.5.3 Techniques utilisées avant la recherche

4.5.3.1 Filtrage et consistance locale

Les techniques de consistance locale tentent de réduire l'espace de recherche avant la résolution. Le filtrage représente l'établissement de la consistance locale. Elles sont étroitement liées à la notion d'hypergraphe associé au CSP. On cherche à supprimer les n-uplets de valeurs qui ne figureront dans aucune solution afin de réduire la complexité totale du problème. Plusieurs propriétés de consistance locale[45] ont été définies dans [74] pour les CSP binaires, telles que la consistance de noeud, d'arc et de

45. on trouve chez certains auteurs le terme de consistance partielle, le terme consistance locale étant réservé à la consistance d'une seule contrainte.

chemin. Ces propriétés sont généralisables aux CSP n-aires en fonction de la définition de la k-consistance :

Définition 4.5.1 - k-consistance - *Un CSP est dit k-consistant si pour tout n-uplet $V \subset \mathcal{V}$ de $k-1$ variables $(x_1,\ldots,x_{k-1})$ et pour toute instanciation consistante $\mathcal{I}$ de ces variables, il existe une valeur $d_k \in D_k$ telle que l'instanciation $\mathcal{I} \cup \{x_k = d_k\}$ soit consistante.*

Autrement dit, il existe un prolongement consistant de V pour toute variable x n'appartenant pas à V. On est alors en mesure d'établir une propriété plus forte que la k-consistance afin de relier la notion de k-consistance à celle de consistance.

Définition 4.5.2 - k-consistance forte - *Un CSP $\mathcal{P}$ est dit fortement k-consistant si et seulement si $\forall i,(1 \leq i \leq k)$ $\mathcal{P}$ est i-consistant.*

Un CSP composé de n variables fortement n-consistant est consistant. Cependant, en raison du coût associé à l'établissement de la k-consistance, on ne recherche en pratique que les consistances de niveau 2 ou 3.

La consistance d'arc[46] [74] est la plus connue et la plus ancienne des consistances locales et correspond dans le cadre des CSP binaires à la 2-consistance forte. Elle fut utilisée pour la première fois par **Waltz** [124, 125] pour résoudre un problème de reconnaissance de formes géométriques.

Son principe repose sur l'identification et la suppression de valeurs n'apparaissant dans aucun des n-uplets d'une contrainte. Un CSP est alors dit arc-consistant s'il ne contient pas de telles valeurs.

Définition 4.5.3 - Consistance d'arc - *Un CSP binaire $\mathcal{P} = \langle \mathcal{V},\mathcal{D},\mathcal{C} \rangle$ est arc-consistant si et seulement si pour toute variable $x_i \in \mathcal{V}$, on a $D_i \neq \emptyset$ et $\forall d_i \in D_i$, $\forall C_j = (V_j,R_j) \in \mathcal{C}$ telle que $x_i \in V_j$, il existe $I_j \in R_j$ tel que $I_j(x_i) = d_i$.*

Une arête de l'hypergraphe est composée de deux arcs. Un arc $(x_i \rightharpoonup x_j)$ est arc-consistant si pour chaque valeur $d_i \in D_i$, il existe une valeur $d_j \in D_j$ telle que l'instanciation $x_i = d_i$, $x_j = d_j$ satisfasse la contrainte existant entre x_i et x_j.

Notons que la notion d'arc-consistance est orientée. En d'autres termes, la consistance de l'arc $(x_i \rightharpoonup x_j)$ n'implique pas forcément la consistance de l'arc $(x_j \rightharpoonup x_i)$.

On préférera travailler, dans le cas de l'arc-consistance, sur l'hypergraphe associé au CSP et faire intervenir la notion d'arc et non plus d'arête, c'est pourquoi on parle effectivement d'arc-consistance. Chaque arête du graphe, que nous notons $(x_i \rightleftharpoons x_j)$, est décomposée en 2 arcs $(x_i \rightharpoonup x_j)$ et $(x_j \rightharpoonup x_i)$.

De nombreux algorithmes capables d'établir la consistance d'arc on été développés et diffèrent suivant leur niveau de complexité. Ces algorithmes reposent sur la fonction *Revise* (cf. figure 4.6), qui étant donné un arc $(x_i \rightharpoonup x_j)$ entre deux variables x_i et x_j, supprime toute valeur du domaine D_i de x_i si elle n'est pas consistante avec une valeur de D_j pour la contrainte (x_i,x_j).

```
fonction Revise( (x_i ⇀ x_j) )
debut
    supval = faux
    pour chaque valeur d_i ∈ D_i faire
        si ∄d_j ∈ D_j satisfaisant (x_i ⇀ x_j) alors
            D_i = D_i \ {d_i}
            supval = vrai
        fsi
    fpour
    retourner supval
fin
```

FIG. 4.6 – *Fonction Révise dans le cadre des CSP binaires.*

46. arc-consistency en anglais.

```
procedure AC1
debut
    soit Arcs l'ensemble des arcs de l'hypergraphe
    repeter
        recommence = faux
        pour chaque (x_i ⇀ x_j) ∈ Arcs faire
            si Revise((x_i ⇀ x_j)) = vrai alors recommence = vrai fsi
        fpour
    tantque recommence = vrai
fin
```

FIG. 4.7 – *Procédure AC-1 dans le cadre des CSP binaires.*

La réduction du domaine d'une variable par la fonction *Revise* peut entraîner l'apparition de nouvelles valeurs arc-inconsistantes sur des variables déjà *résvisées.* Les différents algorithmes d'arc-consistance se distinguent par leur finesse de choix dans la remise en cause des arcs candidats à la révision. Ainsi AC-1 présenté figure 4.7 est le plus primitifs de ces algorithmes et AC-6 [10] le plus performant (voir [1] pp 254–258 pour de plus amples détails).

La consistance de chemin[47] correspond dans le cadre des CSP binaires à la 3-consistance forte.

Définition 4.5.4 - Consistance de chemin - *Un CSP binaire $\mathcal{P} = \langle \mathcal{V},\mathcal{D},\mathcal{C} \rangle$ est chemin-consistant si et seulement si pour tout ensemble de variables $V = \{x_1, \ldots ,x_k\}$ toute instanciation $\mathcal{I}$ de V est consistante, et de plus $\forall i,(1 \leq i \leq k-1)$, l'instanciation de x_{i-1},x_i doit être consistante.*

L'établissement de la consistance de chemin est plus délicate à mettre en oeuvre car elle met en jeu deux contraintes ayant une variable commune. Elle consiste à supprimer pour un couple de variables (x_i,x_j) les couples de valeurs inconsistantes avec une troisième variable x_k. Cette suppression s'effectue par ajout d'une nouvelle contrainte entre x_i et x_j. L'algorithme le plus efficace pour établir la consistance de chemin est PC-4 de complexité temporelle en $O(n^3.d^3)$

L'application d'un filtrage dépend de sa rentabilité. Si un algorithme de filtrage ne supprime que peu de valeurs relativement au coût de son temps de calcul, son application ne se justifie pas. Signalons qu'il peut sembler parfois avantageux de ne pas laisser un algorithme de filtrage aller jusqu'à son terme. Le CSP obtenu n'est pas forcément k-consistant, mais est probablement plus simple que le CSP initial.

Les techniques de filtrage sont en général insuffisantes pour résoudre complétement un problème. Les seuls cas de résolution favorables sont celui où la réduction des domaines conduit à un domaine vide, le problème est alors insatisfiable ; et celui ou tous les domaines sont réduits à un singleton, la solution est alors évidente. Dans les autres cas, il faut procéder à une énumération sur les domaines.

4.5.4 Techniques utilisées durant la recherche

Parmi ces techniques, on trouve les méthodes *rétrospectives* ou **look back** qui, en cas d'échec, tentent de revenir à la cause réelle de l'échec, et les méthodes *prospectives* ou **look ahead** qui après chaque assignation d'une variable réalisent un filtrage des domaines des variables non instanciées afin d'élaguer l'arbre de recherche au plus tôt.

4.5.4.1 Look back

Le *backjumping* [45] consiste, en cas d'inconsistance, à analyser l'instanciation partielle afin de déterminer les assignations responsables de l'échec. Si l'ensemble des valeurs de la variable courante est incompatible avec des assignations précédentes, l'algorithme remet en cause l'assignation la moins récente.

47. path-consistency en anglais.

Exemple L'exemple suivant illustre le *backjumping*:

$$(P_3)\left\{\begin{array}{l} \mathcal{V}=\{x_1,x_2,x_3,x_4\} \\ \mathcal{D}=\left\{\begin{array}{l} D_1=\{1,2\} \\ D_2=\{2,3,4\}, \\ D_3=\{1,2,3\}, \\ D_4=\{1,2,3\} \end{array}\right\} \\ \mathcal{C}=\left\{\begin{array}{ll} (C_1) & x_1<x_2 \\ (C_2) & x_2<x_3 \\ (C_3) & x_1>x_4 \end{array}\right\} \end{array}\right.$$

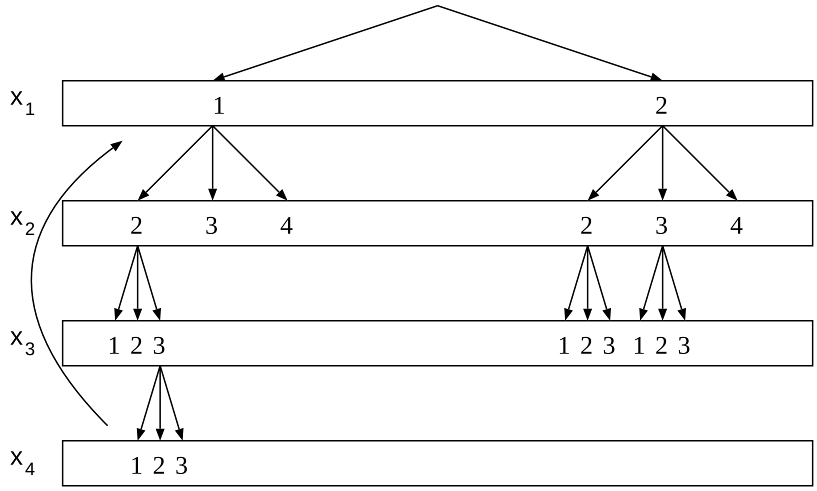

FIG. 4.8 – *Résolution avec BJ*

L'arbre de recherche pour le problème (P_3) est représenté figure 4.8. Lorsque $x_1 = 1$ et $x_2 = 2$, la variable x_3 ne peut prendre que la valeur 3, sinon elle violerait la contrainte C_2. Tentons à présent d'instancier x_4. Toutes les valeurs du domaine D_4 violent la contrainte C_3. Dans le cas d'un backtrack simple, on serait amené à remettre en cause l'instanciation de x_3. Dans le cas du backjumping, la prise en compte des informations issues du viol de la contrainte C_3 permet de déterminer que l'instanciation de x_1 est source du conflit. On effectue donc un retour arrière *non chronologique* directement en x_1, comme indiqué sur la figure.

Le *backjumping dirigé par les conflits*[48] [96] consiste à mémoriser pour chaque variable x_i instanciée, l'ensemble $V_c(x_i)$ des variables rendant inconsistante une des valeurs testée de x_i. En cas d'échec, l'algorithme remet en cause la variable $x_k \in V_c(x_i)$ la plus récemment instanciée et ajoute à l'ensemble $V_c(x_k)$, les variable de $V_c(x_i)$ excepté x_i.

Le *backchecking* [53] cherche à réduire le nombre de tests de consistance. Si l'instanciation d'une variable est inconsistante avec une instanciation précédente on mémorise alors l'inconsistance entre les variables et leurs valeurs respectives. Une amélioration du *backchecking* est le *backmarking* [46].

4.5.4.2 Look ahead

Le *looking ahead* [53] est une expression regroupant sous sa houlette un ensemble de méthodes telles que le *forward checking* (FC), le *partial look-ahead* (PL), le *full look ahead* (FL) et le *really full look ahead* (RFL), chargées d'assurer, de manière plus ou moins prononcée, l'arc-consistance du problème pour l'instanciation courante.

Le *forward checking*, le plus connu des algorithmes prospectifs, assure que les variables non encore instanciées possèdent au moins une valeur consistante avec les variables déjà instanciées. Si l'algorithme

48. conflict based backjumping.

```
fonction FC(X_I,X_N)
entree variables instanciées X_I et non instancièes X_N
debut
    pour chaque variable x_j ∈ X_N faire
        pour chaque x_i ∈ X_I faire
            ok = revise( (x_j ⇀ x_i) )
            si D_j = ∅ alors retourner faux fsi
            fpour
        fpour
        retourner vrai
    fin
```

FIG. 4.9 – *Algorithme du forward checking.*

trouve qu'un domaine d'une variable non instanciée est vide, il remet en cause la valeur de la variable précédemment instanciée (voir figure 4.9).

4.5.5 Techniques utilisées pour guider la recherche

L'étude du comportement algorithmique du retour arrière montre que 2 ordres influencent fortement le temps de résolution des problèmes :

- l'ordre d'instanciation des variables,
- l'ordre d'instanciation des valeurs.

4.5.5.1 Ordre sur les variables

L'ordonnancement des variables (dit ordonnancement *vertical*) peut être soit *statique*, c'est-à-dire déterminé avant la résolution [53, 32] soit *dynamique*, i.e. déterminé durant la résolution [98, 32]. Ces ordonnancements se fondent de manière générale sur le *first-fail principle*[49] qui consiste à lever le plus tôt possible les cas d'inconsistances.

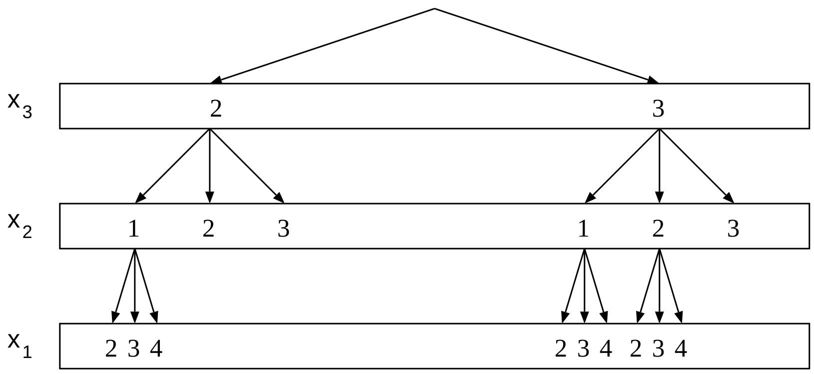

FIG. 4.10 – *Influence de l'ordre d'instanciation des variables.*

Dans le cas statique, le choix de la variable de cardinalité maximum, i.e. la variable connectée par les contraintes au plus grand nombre de variables déjà instanciées, semble être un ordre performant. On peut également choisir les variables par ordre croissant des domaines.

49. que nous pourrions traduire par principe de l'échec d'abord.

4.5.5.2 Ordre sur les valeurs

L'ordre de sélection des valeurs (dit ordre *horizontal*) possède une moindre influence. On peut par exemple choisir de sélectionner les valeurs qui maximisent les cardinaux des domaines des variables non instanciées. Cet ordre se justifie pour des problèmes possédant de nombreuses solutions, on augment ainsi la probabilité de trouver une solution. L'un des ordres les plus connus est l'ordonnancement *min-conflict* [83] qui sélectionne les valeurs suivant l'ordre croissant des conflits qu'elles engendrent au sein des contraintes.

4.6 Méthodes non systématiques

Les méthodes non systématiques, également qualifiées de méthodes de *descente en gradient*[50], sont basées sur le principe de recherche locale. Elles ont été proposées pour résoudre des problèmes d'optimisation combinatoires et se fondent sur la notion de voisinage associée aux configurations.

Définition 4.6.1 - Voisinage - *Deux configurations sont dites voisines si elles ne diffèrent que par la valeur d'une seule variable. Le voisinage d'une configuration représente l'ensemble des configurations voisines pour une configuration donnée.*

Les algorithmes basés sur la recherche locale sont définis par trois critères :

- *le choix d'une configuration initiale*, qui peut être aléatoire ou répondre à certaines propriétés, comme par exemple, satisfaire un nombre minimum de contraintes,
- *le choix d'un critère de déplacement*, basé sur une fonction d'évalutation qui, à chaque configuration dans le voisinage de la configuration courante, associe une valeur (ou facteur de qualité). La *meilleure* configuration voisine sera ensuite choisie jusqu'à l'obtention d'une solution ou d'un blocage,
- *le choix d'un critère de terminsaison.*

Remarquons que les méthodes de recherche locale se fonde sur l'incomplétude. Tout comme pour l'*unit-resulting resolution* en Démonstration Automatique, la recherche locale compense son incomplétude par une abilité à résoudre dans certains cas plus efficacement certaines classes de problèmes. L'incomplétude provient de la notion de recherche de la meilleure configuration candidate pour un critère d'évalutation donné.

Van Hentenryck et **Michel** [120] ont proposé récemment un cadre formel pour la modélisation et le développement d'algorithmes de recherche locale. Le langage de modélisation issu de ces travaux capture les notions de voisinage, de critère d'acceptation et de configuration initiale. Cette approche a pour objectif de diminuer le temps de développement des algorithmes de recherche locale sans affecter leur efficacité.

4.6.1 Descente stochastique

Cette méthode se base sur le choix d'une configuration initiale quelconque. Le critère de déplacement impose que la prochaine configuration diminue le nombre de contraintes insatisfaites. L'algorithme termine si on aboutit à une solution ou si aucune configuration voisine ne satisfait le critère de déplacement.

4.6.2 Algorithme Glouton

L'algorithme Glouton est une variante de la descente stochastique dans laquelle le critère de déplacement se base sur la minimisation d'une fonction objectif définie sur les configurations, ce qui permet de descendre plus rapidement vers un optimum.

Le principal défaut des méthodes précédentes est qu'elles obéissent aveuglément à la fonction objectif. Or celle-ci est basée sur des critères heuristiques qui orientent la recherche dans une direction donnée, ce qui peut conduire l'algorithme dans un voisinage où aucune amélioration de la fonction objectif n'est possible. On parle alors d'*optimum local* à la différence d'un *optimum global* qui représente une solution.

50. hill climbing en anglais.

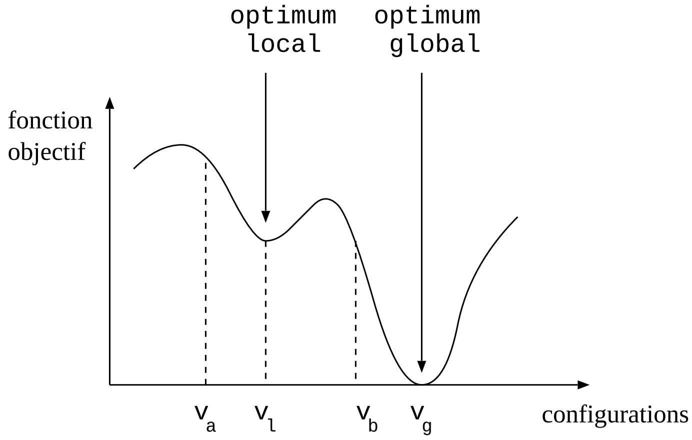

FIG. 4.11 – *Optimum local et optimum global pour une fonction objectif.*

La figure 4.11 illustre ce problème. Partant de v_a, on descendra vers l'optimum local correspondant à v_l où l'on restera bloqué. Par contre, partant de v_2, on atteindra l'optimum global correspondant à la configuration v_g.

Une amélioration triviale pour éviter de rester bloqué dans un optimum local consiste à partir à partir d'une nouvelle configuration initiale.

4.6.3 La méthode Tabou

Cette méthode [51] repose sur l'extension de la notion de voisinage. L'ensemble des voisins ne représentent plus l'ensemble des configurations obtenues par modification de l'instanciation d'une variable, mais se base sur la modification d'*attributs*. Dans le cadre des problèmes booléens, un voisin peut résulter de la modification de plusieurs instanciations ou du changement d'une fonction qui apparaît naturellement dans la formulation du problème. La mémorisation des attributs est alors utilisée pour imposer des contraintes, dites *restrictions tabou*, qui interdisent de revenir à une configuration déjà étudiée en inversant les déplacements réalisés précédemment. Ces restrictions sont *basées sur la proximité* si elles sont limitées à un nombre limités de modifications antérieures, ou *basées sur la fréquence* si elles apparaissent de façon répétitive sur un grand nombre de modifications. Il est cependant possible que les restrictions choisies soient trop fortes et mènent à un bloquage. On a donc été conduit à introduire des *critères d'aspiration* qui ont pour but de déterminer si une restriction peut être relachée (voir [14]). L'algorithme du Tabou est alors fortement dépendant des restrictions et critères d'aspiration choisis par l'utilisateur.

4.6.4 Le recuit simulé

Le recuit simulé a été proposé en Physique/Chimie pour simuler le processus de refroidissement d'un matériau dans un bain chaud[51]. **Metropolis** et al. [82] proposèrent une heuristique capable de simuler le comportement thermodynamique d'un ensemble d'atomes, initialement dans un état d'équilibre à une certaine température, soumis à un refroidissement. Il est bien connu qu'un refroidissement brutal engendre un désordre des atomes qui bloque le système dans un état de haute énergie, alors qu'un refroidissement lent amène le système à un état stable de basse énergie. Le recuit simulé reproduit ce processus de refroidissement lent.

Plus de trente ans après la parution de cette publication, **Kirkpatrick** et al. [59] proposèrent d'appliquer une approche similaire à des problèmes combinatoires d'optimisation. **Cerny** a appliqué cette méthode au problème du voyageur de commerce [19].

Dans cette méthode, une configuration initiale est choisie, puis un changement local est considéré et son effet sur la fonction objectif est évalué. S'il correspond à une amélioration, le changement local est effectué.

51. processus appelé *recuit*

Par contre, si on n'améliore pas la fonction objectif, il s'agit alors d'une détérioration et une probabilité d'acceptation est calculée. Cette probabilité décroît quand la détérioration croît et quand le temps écoulé depuis le début de la simulation croît. Un nombre aléatoire entre 0 et 1 est généré et le changement local est accepté seulement si ce nombre aléatoire est inférieur à la probabilité calculée. La simulation se termine lorsqu'aucun changement local n'a été accepté pendant un nombre donné d'itérations.

4.7 Conclusion

En conclusion de ce chapitre, nous retiendrons que les méthodes de résolution des problèmes de satisfaction de contraintes sont divisées en deux grandes familles : les méthodes systématiques basées sur le retour arrière complètes, constructives et déterministes, et les méthodes non systématiques basées sur la recherche locale, incomplètes et stochastiques. Chacune présente des qualités et des défauts qui contribuent à les rendre plus ou moins efficaces sur des classes de problèmes différentes. En particulier, les méthodes systématiques sont moins efficaces que les méthodes locales sur les problèmes conséquents, mais les méthodes locales péchent par leur incomplétude.

Bien que les approches systématiques et non systématiques soient *orthogonales*, de récents travaux ont conduit à les faire coopérer afin de bénéficier de la complétude des premières et de l'efficacité des secondes. La méthode *Score* [21] est un exemple d'approche hybride fondée sur la génération d'une configuration initiale du problème qui est ensuite répararée (aspect non systématique) en utilisant un mécanisme de mémorisation des configurations testées fondé sur le backtrack (aspect systématique). Ce mécanisme permet d'éviter de rester bloqué dans un optimum local. L'utilisation à chaque instant d'une configuration globale du problème influe sur l'efficacité de sa résolution notamment lorsque l'on utilise une heuristique du type *Min-Conflicts* [83].

Deuxième partie

Contribution

Chapitre 5

CSP et Calcul propositionnel

Dans ce chapitre nous exhibons, dans un premier temps, la puissance de l'opérateur de cardinalité transposé au calcul propositionnel, tant au niveau modélisation que déduction et montrons qu'il permet d'établir un système de déduction qui traite les problèmes de manière globale. Nous introduisons également des notions en anticipation de ce que nous verrons dans le prochain chapitre.

5.1 Modélisation

Parmi les différentes méthodes élaborées pour résoudre les problèmes du calcul propositionnel, les Problèmes de Satisfaction de Contraintes offrent un attrait particulier car ils privilégient la modélisation, correspondant au point de vue de l'utilisateur, par rapport à la résolution, point de vue du concepteur. Les techniques de résolution associées aux CSP se sont montrées remarquablement efficaces dans le cas général pour résoudre les problèmes du calcul propositionnel ([21]).

Le problème de satisfiabilité (SAT) consiste à déterminer s'il existe une instanciation des variables propositionnelles d'un ensemble de clauses C de manière à ce que C soit satisfiable. Parmi les algorithmes conçus pour résoudre le problème SAT, les méthodes se fondant sur l'utilisation de techniques CSP ont prouvé qu'elles étaient plus efficaces (voir [109]) que la procédure de **Davis** et **Putnam** [31]. Ces méthodes sont susceptibles d'être classées en deux catégories : les approches systématiques comme C-SAT [37], *Score* [21] qui sont complètes, et les approches non systématiques comme GSAT [109] qui sont incomplètes.

La transformation d'un problème du calcul propositionnel, exprimé sous forme normale conjonctive, en CSP est relativement simple. Les variables propositionnelles deviennent les variables du CSP et sont généralement définies sur le domaine fini bi-valué $\{0,1\}$ ou le domaine booléen $\{vrai,faux\}$, la valeur 1 correspond à la valeur de vérité $vrai$, et 0 à $faux$. Les clauses sont facilement représentables sous forme de *clauses de cardinalité*, dérivées de l'*opérateur de cardinalité* [122], qui allient concision et un pouvoir expressif accru.

Définition 5.1.1 - Opérateur de cardinalité - *On note* $\#< \alpha,\beta,C_1,\ldots,C_n >$*, l'opérateur de cardinalité où* α,β *sont des entiers, et* $C_1,\ldots,C_n$ *des contraintes. L'opérateur de cardinalité est une contrainte satisfaite si au moins* α *et au plus* β *contraintes de* $C_1,\ldots,C_n$*, , sont satisfaites (avec* $0 \leq \alpha \leq \beta \leq n$*).*

Nous utilisons par la suite une version spécifique de l'opérateur de cardinalité dédiée aux booléens et qualifiée de *clause de cardinalité* :

Définition 5.1.2 - Clause de cardinalité - *On note* $\#< \alpha,\beta,p_1,\ldots,p_n >$*, la clause (ou contrainte) de cardinalité satisfaite si au moins* α *et au plus* β *littéraux de* $p_1,\ldots,p_n$*, sont interprétés à* vrai *(avec* $0 \leq \alpha \leq \beta \leq n$*).*

Proposition 5.1.1 - Equivalence clause CNF/clause de cardinalité - *Toute clause sous forme normale conjonctive* $p_1 \vee p_2 \vee \ldots \vee p_n$ *est équivalente à une clause de cardinalité de la forme :*

$$\#< 1,n,p_1,p_2,\ldots,p_n >$$

L'avantage des clauses de cardinalité par rapport aux clauses sous forme normale conjonctive réside dans leur concision (voir table 5.1 tirée de [58]). Un ensemble de clauses CNF peut être remplacé avantageusement par une seule clause de cardinalité, ce qui a pour effet de diminuer la complexité du problème à résoudre.

Clause de cardinalité	Equivalent CNF
$\# < \alpha, \beta, p_1, p_2, \ldots, p_n >$ $0 < \alpha \leq \beta < n$	$C_n^{n-\alpha+1}$ clauses : $p_{i_1} \vee \ldots \vee p_{i_{n-\alpha+1}}$ avec $\{p_{i_1}, \ldots, p_{i_{n-\alpha+1}}\} \subset \{p_1, \ldots, p_n\}$ $C_n^{\beta+1}$ clauses : $\neg p_{i_1} \vee \ldots \vee \neg p_{i_{\beta+1}}$ avec $\{p_{i_1}, \ldots, p_{i_{\beta+1}}\} \subset \{p_1, \ldots, p_n\}$
$\# < \alpha, n, p_1, p_2, \ldots, p_n >$ $0 < \alpha \leq n$	$C_n^{n-\alpha+1}$ clauses : $p_{i_1} \vee \ldots \vee p_{i_{n-\alpha+1}}$ avec $\{p_{i_1}, \ldots, p_{i_{n-\alpha+1}}\} \subset \{p_1, \ldots, p_n\}$
$\# < 0, \beta, p_1, p_2, \ldots, p_n >$ $0 \leq \beta < n$	$C_n^{\beta+1}$ clauses : $\neg p_{i_1} \vee \ldots \vee \neg p_{i_{\beta+1}}$ avec $\{p_{i_1}, \ldots, p_{i_{beta+1}}\} \subset \{p_1, \ldots, p_n\}$

TAB. 5.1 – *Concision des clauses de cardinalité*

Le cas de figure le plus significatif correspond au cas où $\alpha = \beta = 1$. Une seule clause de cardinalité composée de n variables propositionnelles remplace $\frac{n \times (n-1)}{2} + 1$ clauses sous forme normale conjonctive.

$$\# < 1, 1, p_1, p_2, \ldots, p_n > \quad \Longleftrightarrow \quad \begin{cases} p_1 \vee p_2 \vee \ldots \vee p_n \\ \neg p_1 \vee \neg p_2, \ldots, \neg p_1 \vee \neg p_n \\ \vdots \\ \neg p_{n-1} \vee \neg p_n \end{cases}$$

5.2 Transformation en CSP

Nous définissons à présent une procédure de transformation des problèmes du calcul propositionnel écrits sous forme normale conjonctive en problèmes de satisfaction de contraintes en domaine **fini booléen**, modélisés sous forme de clauses de cardinalité. Cette procédure est linéaire en la taille des données du problème initial.

Définition 5.2.1 - Transformation Tr_0 **-** *Tout problème du calcul propositionnel sous forme normale conjonctive $P = \{c_1, \ldots, c_m\}$, composé de m clauses et n variables propositionnelles $p_1, \ldots, p_n$, est modélisable sous forme d'un Problème de Satisfaction de Contraintes sur le domaine booléen $B = \{vrai, faux\}$, $CSP^B\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, par application des règles suivantes :*

- *$\mathcal{V} = \{p_1, \ldots, p_n\}$ est composé des variables propositionnelles de P,*
- *$\mathcal{D} = \{d_1, \ldots, d_n\}$ est composé des domaines booléens $\{vrai, faux\}$ associés aux variables de $\mathcal{V}$,*
- *$\mathcal{C} = \{c_1, \ldots, c_m\}$ est composé de m contraintes de cardinalités $\# < 1, k_i, \mathcal{L}_i >$, une pour chaque clause c_i de P, de longueur k_i. $\mathcal{L}_i$ représente l'ensemble des littéraux de la clause c_i.*

Proposition 5.2.1 - *Tr_0 est une transformation polynomiale.*

Preuve En effet la construction du domaine $\mathcal{V}$ se fait en n étapes, ainsi que celle de $\mathcal{D}$. La traduction des contraintes est réalisée en m étapes, soit au total $2n + m$ étapes.

Exemple : Soit le problème suivant :

$$P_1 \begin{cases} (c_1) & p_1 \vee p_2 \vee \neg p_3 \\ (c_2) & p_4 \vee \neg p_5 \\ (c_3) & p_2 \vee p_3 \end{cases}$$

Le problème P_1 se réécrit par Tr_0 en :

$$CSP^B\mathcal{P}_1 \left\{ \begin{array}{ll} \mathcal{V} = & \{p_1,\dots p_5\} \\ \mathcal{D} = & \{d_1,\dots d_5\} \text{ avec } d_i = \{vrai,faux\}, \forall i \in \{1..5\} \\ \mathcal{C} = & \left\{ \begin{array}{l} \#< 1,3,p_1,p_2,\neg p_3 > \\ \#< 1,2,p_4,\neg p_5 > \\ \#< 1,2,p_2,p_3 > \end{array} \right\} \end{array} \right.$$

5.3 Système de déduction

L'opérateur de cardinalité se prête à l'élaboration d'un système de déduction élégant et efficace pour les problèmes structurés (comme le problème des Pigeons), dont le but est d'établir l'insatisfiabilité d'une formule du calcul propositionnel exprimée sous forme d'un ensemble de clauses de cardinalité.

Ce système, que nous avons baptisé Ω, représente une théorie axiomatique, faisant appel à l'arithmétique sur les entiers naturels et dont les axiomes sont des clauses de cardinalité. Les deux principales règles d'inférence de ce système sont la règle d'*Extension* et d'*Absorption*.

La règle d'*Extension* peut être considérée comme l'application du *principe de résolution* sur deux clauses (ou contraintes), sans que les littéraux complémentaires ne soient éliminés, rôle dévolu à la règle d'*Absorption*. Ce système reprend en somme l'idée de l'élimination de modèle (voir section 3.4.2). L'action conjuguée des règles *Extension/Absorption*, conduit dans certains cas, à des représentations *canoniques* qui correspondent à des cas triviaux d'instanciation d'une ou d'un ensemble de variables. Une inconsistance survient dès lors que deux formes canoniques forcent une variable propositionnelle à être instanciée à des valeurs contradictoires.

Le principal intérêt de Ω réside dans sa potentialité à traiter les problèmes de façon globale. Son élaboration ne repose pas sur un cadre théorique précis, mais sur une utilisation pragmatique de l'opérateur de cardinalité. Dans son principe, le système Ω combine les clauses de cardinalité (règle d'*Extension*) pour ensuite réduire la taille des nouvelles clauses obtenues (règle d'*Absorption*). L'application des règles d'inférence de Ω sur un problème P permet :

- soit d'aboutir à l'inconsistance du problème ($P \xrightarrow{\Omega} \perp$),
- soit de déterminer une instanciation I d'un sous-ensemble de variables ($P \xrightarrow{\Omega} I$),
- soit dans le pire des cas, de ne rien déduire ($P \xrightarrow{\Omega} \emptyset$).

5.4 Définition formelle

Nous donnons à présent une description formelle de Ω.

Définition 5.4.1 - Alphabet - *L'alphabet de Ω est composé des symboles suivants :*

- *$\alpha,\beta,\alpha_1,\beta_1,\dots,\alpha_n,\beta_n \in N$ sont des entiers positifs ou nuls, obéissant aux lois de l'arithmétique de* **Peano**,
- *p est un littéral,*
- *$\mathcal{L},\mathcal{L}_1,\dots,\mathcal{L}_n$ sont des listes de littéraux, $\emptyset$ représente une liste vide et $|\mathcal{L}|$ la taille de la liste, c'est-à-dire le nombre de littéraux de la liste,*
- *vrai, faux représentent des constantes booléennes, telles que $\neg vrai = faux$ et $\neg faux = vrai$ ($\neg$ représente la négation logique),*
- *ϵ remplace une valeur de vérité quelconque $\epsilon \in \{vrai,faux\}$,*
- *$\perp$ symbolise l'inconsistance.*

Définition 5.4.2 - Formules bien formées - *Une formule bien formée de Ω peut prendre la forme*

- *d'une clause de cardinalité : $\#< \alpha,\beta,\mathcal{L} >$*
- *ou d'une instanciation de variables : $\mathcal{L} = \epsilon$, $\epsilon \in \{vrai,faux\}$.*

Remarque : La formule d'instanciation n'est pas une instanciation au sens sémantique du terme, mais représente bien un mot du langage.

Définition 5.4.3 - Relation de conséquence - *On note $\vdash_\Omega$ la relation de conséquence ([43], p. 179 - 216) satisfaisant les propriétés suivantes :*

- **Réflexivité**

$$\Gamma \vdash_\Omega \Gamma$$

- **Monotonie** [52]

$$\frac{\Gamma \vdash_\Omega \Delta}{\Gamma, A \vdash_\Omega \Delta}$$

- **Transitivité**

$$\frac{\Gamma \vdash_\Omega \Phi \qquad \Phi \vdash_\Omega \Delta}{\Gamma \vdash_\Omega \Delta}$$

Dans la suite de ce chapitre et afin d'alléger les notations, nous noterons $\vdash$ au lieu de $\vdash_\Omega$.

Définition 5.4.4 - Axiomes - *Les axiomes de Ω sont les opérateurs de cardinalité spécifiant le problème. Pour toute clause de cardinalité $c \in CSP^B\mathcal{P}$, on a donc $\vdash c$.*

Définition 5.4.5 - Règles d'inférence -

- **Extension -** La règle d'extension permet de réunir plusieurs contraintes de cardinalité en une seule contrainte afin d'aboutir à un représentant unique :

$$(Ext) \qquad n \geq 2 \qquad \frac{\Gamma_1 \vdash \# < \alpha_1, \beta_1, \mathcal{L}_1 > \ldots \Gamma_n \vdash \# < \alpha_n, \beta_n, \mathcal{L}_n >}{\begin{array}{l}\Gamma_1, \ldots, \Gamma_n \vdash \\ \# < \alpha_1 + \ldots + \alpha_n, \beta_1 + \ldots + \beta_n, \mathcal{L}_1, \ldots, \mathcal{L}_n >\end{array}}$$

- **Absorption -** La règle d'absorbption élimine les littéraux complémentaires :

$$(Abs) \qquad \alpha > 0 \qquad \frac{\Gamma \vdash \# < \alpha, \beta, \mathcal{L}, p, \neg p >}{\Gamma \vdash \# < \alpha - 1, \beta - 1, \mathcal{L} >}$$

- **Limitation -** Les règles de limitation rétablissent la cohérence des contraintes après application de la règle d'absorption, elles sont généralement implicites et n'apparaîssent pas lors de la preuve :

$$(Min) \qquad \alpha < 0 \qquad \frac{\Gamma \vdash \# < \alpha, \beta, \mathcal{L} >}{\Gamma \vdash \# < 0, \beta, \mathcal{L} >}$$

$$(Max) \qquad \beta > |\mathcal{L}| \qquad \frac{\Gamma \vdash \# < \alpha, \beta, \mathcal{L} >}{\Gamma \vdash \# < \alpha, |\mathcal{L}|, \mathcal{L} >}$$

- **Instanciation -** Les règles d'instanciation permettent le passage de contraintes de cardinalité qui correspondent à des cas triviaux d'instanciation, à une formule d'instanciation qui termine la recherche pour la branche courante de la preuve :

52. parfois appelée Dilution

(Ins_f) $$\frac{\Gamma \vdash \#< 0,0,\mathcal{L} >}{\Gamma \vdash \mathcal{L} = faux}$$

(Ins_v) $|\mathcal{L}| = n$ $$\frac{\Gamma \vdash \#< n,n,\mathcal{L} >}{\Gamma \vdash \mathcal{L} = vrai}$$

(Ins_d) $\beta \geq 1$ $$\frac{\Gamma \vdash \#< 1,\beta,p,p >}{\Gamma \vdash p = 1}$$

- **Inconsistance -** Les règles d'inconsistance terminent la résolution du problème en démontrant son inconsistance :

(Inc_1) $[\alpha_1,\beta_1] \cap [\alpha_2,\beta_2] = \emptyset$ $$\frac{\Gamma_1 \vdash \#< \alpha_1,\beta_1,\mathcal{L} > \qquad \Gamma_2 \vdash \#< \alpha_2,\beta_2,\mathcal{L} >}{\Gamma_1,\Gamma_2 \vdash \bot}$$

(Inc_2) $$\frac{\Gamma_1 \vdash \mathcal{L}_1,p = \epsilon \qquad \Gamma_2 \vdash \mathcal{L}_2,\neg p = \epsilon}{\Gamma_1,\Gamma_2 \vdash \bot}$$

(Inc_3) $$\frac{\Gamma_1 \vdash \mathcal{L}_1,p = \epsilon \qquad \Gamma_2 \vdash \mathcal{L}_2,p = \neg\epsilon}{\Gamma_1,\Gamma_2 \vdash \bot}$$

- **Impasse -** Cette règle termine le développement d'une branche de la preuve dans le cas où la contrainte ne contient plus de littéraux :

(Imp) $\beta \geq 0$ $$\frac{\Gamma \vdash \#< 0,\beta,\emptyset >}{\Gamma \vdash}$$

5.5 Propriétés de Ω

5.5.1 Terminaison

Le système Ω termine si l'on convient de ne pas appliquer la règle d'*Extension* entre une clause de cardinalité c et une clause résultant de l'application de la règle d'*Extension* entre c et une autre clause.

Preuve Dans ce cas, la plus grande clause que l'on puisse obtenir est la clause résultant de l'application de la règle d'*Extension* entre toutes les clauses du problème.

5.5.2 Correction

Le système Ω est correct, en ce sens que si, pour un problème P, on a $P \overset{\Omega}{\to} \bot$, alors on est capable de prouver l'inconsistance de P par application du *principe de résolution*.

Preuve Comme nous l'avons précisé précédemment, l'utilisation conjointe des règles d'*Extension/Absorption* possède le même comportement que le *principe de résolution* et Ω se conforme, de ce point de vue, aux principes de base de la logique. Si l'on parvient à démontrer l'inconsistance de P par Ω, cela signifie :

– soit qu'il existe deux instanciations contradictoires (règles Inc_2,Inc_3) d'une variable p ; ce qui, au regard du *principe de résolution*, revient à démontrer p et $\neg p$,

– soit qu'une incompatibilité entre clauses a été levée (règle Inc_1) ; cette propriété, spécifique aux clauses de cardinalité, ne possède pas d'équivalent en logique classique. Elle est intéressante car elle représente un cas d'inconsistance, pour lequel aucune instanciation n'est nécessaire.

5.5.3 Validité et Incomplétude

Le système Ω est valide mais malheureusement incomplet (cf. section suivante).

5.5.4 Exemples

Bien que le système Ω ne soit pas complet, il s'avère cependant d'un intérêt non négligeable, puisqu'il permet de résoudre **efficacement** certains types de problèmes.

5.5.4.1 Exemple 1

Considérons le problème suivant :

$$P_2 \left\{ \left\{ \begin{array}{l} \neg p \vee \neg q \vee \neg r \vee s \\ \neg p \vee q \vee r \vee \neg s \\ p \vee \neg q \vee r \vee \neg s \\ p \vee q \vee \neg r \vee \neg s \\ p \vee q \vee r \vee s \\ \neg p \vee \neg q \vee r \vee \neg s \\ \neg p \vee q \vee \neg r \vee \neg s \\ \neg p \vee q \vee r \vee s \\ p \vee \neg q \vee \neg r \vee \neg s \\ p \vee \neg q \vee r \vee s \\ p \vee q \vee \neg r \vee s \\ \neg p \vee \neg q \vee \neg r \vee \neg s \\ \neg p \vee \neg q \vee r \vee s \\ \neg p \vee q \vee \neg r \vee s \end{array} \right. \quad \left\{ \begin{array}{l} \neg p \vee q \vee \neg r \\ p \vee \neg q \vee r \end{array} \right. \quad \left\{ \begin{array}{l} \neg q \vee r \\ q \vee \neg r \end{array} \right. \quad \left\{ \begin{array}{l} q \vee \neg r \vee \neg s \\ \neg q \vee \neg r \vee \neg s \\ q \vee r \vee \neg s \\ q \vee \neg r \vee s \\ \neg q \vee r \vee s \end{array} \right. \quad \left\{ \begin{array}{l} p \vee q \vee r \\ \neg p \vee \neg q \vee \neg r \end{array} \right. \right.$$

dont l'expression sous forme de CSP se ramène simplement à :

$$CSP^B\mathcal{P}_2 \left\{ \begin{array}{ll} \mathcal{V} = & \{p,q,r,s\} \\ \mathcal{D} = & \{d_p,d_q,d_r,d_s = \{vrai,faux\}\} \\ \mathcal{C} = & \left\{ \begin{array}{l} \# < 0,0,p,q,r,\neg s > \\ \# < 1,2,\neg p,q,\neg r > \\ \# < 1,1,\neg q,r > \\ \# < 2,2,q,\neg r,\neg s > \\ \# < 1,2,p,q,r > \end{array} \right. \end{array} \right.$$

L'application des règles de Ω sur ce problème, mène à :

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\# < 2,2,q,\neg r,\neg s > \quad \# < 1,1,\neg q,r >}{\# < 3,3,q,\neg q,r,\neg r,\neg s >}\ (Ext)}{\# < 2,2,r,\neg r,\neg s >}\ (Abs)}{\# < 1,1,\neg s >}\ (Abs)}{\neg s = vrai}\ (Ins_v) \qquad \cfrac{\# < 0,0,p,q,r,\neg s >}{\ldots,\neg s = faux}\ (Ins_f)}{\perp}\ (Inc_3)$$

On en déduit donc que le problème est inconsistant.

5.5.4.2 Exemple 2 : le problème des pigeons

Ce problème consiste à déterminer s'il est possible de placer n pigeons dans m pigeonniers, un pigeonnier ne pouvant être occupé au plus que par un seul pigeon. Pour modéliser ce problème, on utilise $n \times m$ variables propositionnelles $p_{i,j}$ avec $i \in \{1,n\}$, $j \in \{1,m\}$. Si $p_{i,j}$ est *vraie* alors le pigeon i se trouve dans le pigeonnier j.

Ce problème, archétype des problèmes combinatoires, est intéressant pour plusieurs raisons :

- sa solution est évidente,
- si $n \leq m$, le problème est **consistant** et la recherche d'une solution se fait de manière linéaire sachant que le nombre de solutions (C_m^n) est exponentiel,
- si $n > m$, le problème est **inconsistant** mais les algorithmes standard (du type Davis et Putnam) prouvent l'inconsistance en un temps exponentiel.

Le problème, écrit en fonction des clauses de cardinalité, s'exprime par :

$$\begin{array}{c} \text{chaque pigeon occupe un seul pigeonnier} \\ \text{(soit } n \text{ clauses)} \end{array} \quad \left\{ \begin{array}{l} \# < 1, 1, p_{1,1}, \ldots, p_{1,m} > \\ \ldots \\ \# < 1, 1, p_{n,1}, \ldots, p_{n,m} > \end{array} \right.$$

$$\begin{array}{c} \text{chaque pigeonnier contient au plus un pigeon} \\ \text{(soit } m \text{ clauses)} \end{array} \quad \left\{ \begin{array}{l} \# < 0, 1, p_{1,1}, \ldots, p_{n,1} > \\ \ldots \\ \# < 0, 1, p_{1,m}, \ldots, p_{n,m} > \end{array} \right.$$

Notons, ici, que l'expression du problème est linéaire en la taille des données, puisque le problème s'exprime en $n + m$ clauses de cardinalité, alors que sous forme CNF, elle nécessiterait :

$$n \times (C_m^m + C_m^2) + m \times (C_n^2) \quad \text{clauses CNF (cf. table 5.1)}$$

soit

$$\boxed{n \times \left(1 + \frac{m \times (m-1)}{2}\right) + m \times \left(\frac{n \times (n-1)}{2}\right)} \quad \text{clauses CNF}$$

La résolution de ce problème par le système Ω conduit à :

$$\frac{\# < 1, 1, p_{1,1}, \ldots, p_{1,m} > \ \ldots \ \# < 1, 1, p_{n,1}, \ldots, p_{n,m} >}{\# < n, n, \mathcal{L} >} \ (Ext)$$

$$\frac{\# < 0, 1, p_{1,1}, \ldots, p_{n,1} > \ \ldots \ \# < 0, 1, p_{1,m}, \ldots, p_{n,m} >}{\# < 0, m, \mathcal{L} >} \ (Ext)$$

$$\frac{\# < n, n, \mathcal{L} > \quad \# < 0, m, \mathcal{L} >}{\bot \quad \text{si} \quad n > m} \ (Inc_1)$$

où $\mathcal{L}$ représente l'ensemble des littéraux du problème.

Nous sommes donc à même, grâce au système Ω, de déterminer la consistance ou l'inconsistance du problème des pigeons en $n + m + 1$ étapes, soit en un temps **linéaire** en la taille des données, ce qui représente un gain considérable par rapport à une résolution classique.

5.5.4.3 Exemple 3

Nous donnons à présent un exemple pour lequel, la résolution suivant Ω ne permet pas de déduire directement l'inconsistance du problème. L'exemple choisi correspond à un cas trivial d'inconsistance.

$$CSP^B\mathcal{P}_3 \left\{ \begin{array}{ll} \mathcal{V} = & \{p,q\} \\ \mathcal{D} = & \{d_p,d_q = \{vrai,faux\}\} \\ & \#< 1,2,p,q > \\ \mathcal{C} = & \#< 1,2,\neg p,\neg q > \\ & \#< 1,2,\neg p,q > \\ & \#< 1,2,p,\neg q > \end{array} \right. \quad \Longleftrightarrow \quad \begin{array}{l} p \vee q \\ \neg p \vee \neg q \\ \neg p \vee q \\ p \vee \neg q \end{array}$$

sa résolution par Ω peut mener soit à une impasse :

$$\cfrac{\cfrac{\cfrac{\#< 1,2,p,q > \quad \#< 1,2,\neg p,\neg q >}{\#< 2,4,p,\neg p,q,\neg q >}\,(Abs)}{\#< 1,3,p,\neg p >}\,(Abs)}{\#< 0,2,\emptyset >}\,(Imp)$$

soit à une forme canonique :

$$\cfrac{\cfrac{\cfrac{\#< 1,2,p,q > \quad \#< 1,2,\neg p,q >}{\#< 2,4,p,\neg p,q,q >}\,(Ext)}{\#< 1,3,q,q >}\,(Abs)}{q = vrai}\,(Ins_d)$$

mais ne permet pas de déterminer l'inconsistance du problème.

5.5.4.4 Exemple 4

Pour les problèmes de représentation des connaissances du type problème du logicien de Lewis **Caroll**, le système de déduction Ω ne permet pas d'attribuer une valeur de vérité à un sous-ensemble de littéraux. Il est alors nécessaire de recourir à un solveur qui procède par exemple par saturation. Notons que pour ce genre de problème, l'aspect déductif est prédominant. Il ne s'agit pas tant de déterminer la satisfiabilité (ou consistance) de l'ensemble de clauses, mais plutôt de déterminer les conséquences logiques du problème. L'utilisation d'un démonstrateur est donc préférable. L'énoncé de ce problème est fourni en annexe.

5.5.5 Réduction

Dans le cas où Ω termine sur une formule d'instanciation, du type $p_1,\ldots,p_n = \epsilon$, on peut réduire le problème initial par le procédé décrit ci-après :

Définition 5.5.1 *Un problème de satisfaction de contraintes $CSP^B\mathcal{P}$, stipulé sous forme de clauses de cardinalité, se réduit en un problème $CSP^B\mathcal{P}'$ suivant l'instanciation I par application des règles suivantes :*

pour chaque variable $p_i \in I$ et pour toute clause $c = \#< \alpha,\beta,\mathcal{L},p_i >$, on a :

si	*et si*	*alors c se réécrit en*
$\alpha \geq 1$	$p_i = vrai$	$\#< \alpha - 1,\beta - 1,\mathcal{L} >$
$\alpha \geq 1$	$p_i = faux$	$\#< min(\alpha,\|\mathcal{L}\|),min(\beta,\|\mathcal{L}\|),\mathcal{L} >$
$\alpha = 0$	$p_i = vrai$	$\#< 0,max(0,\beta - 1),\mathcal{L} >$
$\alpha = 0$	$p_i = faux$	$\#< 0,min(\beta,\|\mathcal{L}\|),\mathcal{L} >$

Appliquée à l'exemple précédent, la réduction par l'instanciation $I = \{q = vrai\}$ autorise une simplification du problème en :

$$CSP^B\mathcal{P}_3{}' \left\{ \begin{array}{ll} \mathcal{V} = & \{p,q\} \\ \mathcal{D} = & \{d_p,d_q = \{vrai,faux\}\} \\ & \#< 0,1,p > \\ \mathcal{C} = & \#< 1,1,\neg p > \\ & \#< 0,1,\neg p > \\ & \#< 1,1,p > \end{array} \right.$$

Une nouvelle résolution de ce sous-problème, par Ω mène à l'inconsistance :

$$\frac{\dfrac{\#< 1,1,p >}{p = vrai}(Ins_v) \quad \dfrac{\#< 1,1,\neg p >}{\neg p = vrai}(Ins_v)}{\perp}(Inc_2)$$

5.5.6 Algorithme

A partir des règles de Ω, on peut élaborer un algorithme général de recherche de consistance des problèmes faisant appel à Ω en tant que système de simplification, afin de tenter de briser la complexité du problème initial :

```
fonction  ΩConsistance(P : CSP^B) : booléen
entree : un CSP^B
retourne : vrai si le problème est consistant,
           faux sinon
debut
    {filtrage}
    CSP^B Q = P
    repeter
        I = ΩRésolution(Q)
        si (I ≠ ⊥) et (I ≠ ∅) alors Q = Réduction(Q,I) fsi
    jusqu'a (I = ⊥) ou (I = ∅)

    si tous les domaines sont des singletons alors
        retourne Q
    sinon
        {labeling}
        si (I ≠ ⊥) alors
            instancier les variables suivant I : Q -I-> Q'
            retourne résolution(Q')
        sinon
            retourne résolution(Q)
        fsi
    fsi
fin
```

TAB. 5.2 – *Recherche de consistance par Ω*

Dans un premier temps, on tente de simplifier le problème initial, en fonction des instanciations qui résultent de résolutions successives par Ω. On applique ensuite une méthode de résolution classique pour achever de résoudre le problème simplifié et instancier les variables qui ne le sont pas.

L'étude comportementale de cet algorithme, ramenée au cadre de travail des CSP fait apparaître distinctement deux phases :

- la première, bien que syntaxique et qui correspond à la résolution du problème par Ω peut être considérée comme une étape de **filtrage**, dans laquelle les valeurs inconsistantes sont éliminées par construction d'une instanciation consistante (fonction Ω*Résolution*). L'instanciation obtenue est alors utilisée pour simplifier le problème (fonction *Réduire*),

- dans la seconde phase, soit on a pu démontrer l'inconsistance du problème qui est alors résolu, soit on termine sur une instanciation (éventuellement vide) des variables. Si l'instanciation est totale, en d'autres termes, si le domaine de chaque variable est réduit à un singleton, le problème est consistant. Sinon, l'ensemble des variables du problème est partitionné en deux sous-ensembles : celui des variables instanciées par Ω et celui des variables non instanciées. On procède alors en quelque sorte à une phase de **labeling** des variables non instanciées par appel de la fonction *Résolution*, qui correspond à une procédure de recherche de satisfiabilité classique.

Il existe cependant de nombreux cas pour lesquels Ω est incapable de déduire la valeur de vérité d'un ensemble de variables. Il faut alors faire appel à un serveur externe pour résoudre le problème. Nous préconisons l'emploi de Ω pour des problèmes *structurés* tels *Ramsey*, *Schur*, *Pigeons* pour lesquels l'expression du problème force certaines variables à prendre une valeur de vérité définie.

5.6 Conclusion

Dans ce chapitre, nous nous sommes familiarisés avec les notions qui nous seront utiles par la suite. Nous avons montré qu'une modélisation des problèmes du calcul propositionnel par le biais de l'opérateur de cardinalité était significativement intéressante puisqu'elle apportait une concision dans l'expression des problèmes ainsi qu'une amélioration de la résolution de certaines classes de problèmes par utilisation d'un système de déduction basé sur les clauses de cardinalité.

Chapitre 6

La méthode *Sacre*

Nous nous intéressons dorénavant à la résolution des problèmes du calcul des prédicats et tentons de transposer à l'ordre supérieur les notions entrevues dans le chapitre précédent. Nous montrons que la difficulté d'adaptation de ces méthodes nous contraint à modifier le formalisme CSP afin d'introduire de nouveaux concepts.

6.1 Objectifs

L'objectif de notre travail consiste à définir une nouvelle approche de résolution pour la Démonstration Automatique et de contribuer à l'amélioration de l'efficacité du traitement des problèmes du calcul des prédicats en ayant recours à des techniques de CSP qui ont prouvé leur efficacité en calcul propositionnel.

A l'origine, ce travail se voulait être une extension de la méthode *Score* [101, 58] au calcul des prédicats [53].

Pour atteindre ce but nous devions donc établir :

- un procédé de modélisation des problèmes du calcul des prédicats, exprimés sous forme normale conjonctive, en problèmes de satisfaction de contraintes ;
- un algorithme de résolution des problèmes ainsi obtenus basé sur le formalisme précédent, susceptible de rendre compte de la consistance du problème logique ;

6.1.1 Modélisation

Tout comme en calcul propositionnel, nous définissons une transformation d'un problème de la logique des prédicats en Problème de Satisfaction de Contraintes. Cette transformation introduit un nouveau domaine pour les CSP.

L'idée qui a aiguillé notre approche repose sur une modification de la signification d'une clause unitaire. Soit une clause unitaire $homme(socrate)$. Lorsque nous écrivons $homme(socrate)$ nous voulons signifier que le symbole de constante $socrate$ vérifie la propriété $homme$ ou encore que $homme$ s'applique à $socrate$. De manière identique, la clause $\neg homme(tweety)$ a pour signification : $tweety$ n'est pas un $homme$, ou encore, $homme$ ne s'applique pas à $tweety$.

Maintenant, si nous modifions légèrement la signification donnée aux clauses précédentes et revenons aux sources de la logique des prédicats, i.e. le calcul des classes, nous sommes à même de concevoir les littéraux $homme$ et $\neg homme$ comme la représentation de deux sous-ensembles disjoints, que nous notons $homme^+$ et $homme^-$, qui sont l'émanation d'un même concept (celui d'homme) et qui définissent une répartition des objets : l'ensemble des objets qui vérifient la propriété (ou qui sont des) $homme(s)$, et l'ensemble des objets qui ne sont pas des $homme(s)$. Ainsi, lorsque nous écrivons $homme(socrate)$ (respectivement $\neg homme(tweety)$), nous entendons formuler $socrate \in homme^+$ (respectivement $tweety \in homme^-$). Cette séparation en deux sous-ensembles est également compatible avec

53. Nous ne parlerons pas de *Score* puisque l'approche que nous envisageons est différente des concepts utilisés en *Score*.

la notion d'interprétation des littéraux : *homme+* correspond à l'ensemble des objets vrais dans une interprétation.

Afin de pouvoir formuler les problèmes sous forme CNF en Problème de Satisfaction de Contraintes, nous introduisons la notion de Ψ-variable et Ψ-domaine. Ces notations sont la mise en œuvre de l'idée que nous venons d'évoquer. Une Ψ-variable est un mode de représentation des littéraux sous forme de variable.

Définition 6.1.1 - Ψ-variable - *Soit p un symbole de prédicat. On appelle Ψ-variable (ou variable) la forme dénotationnelle Ψ_p. En particulier, si $p(t)$ est un littéral, où p est de la forme p ou $\neg p$, et t une liste de termes, on note $\Psi_p^{[t]}$, l'occurrence de la Ψ-variable Ψ_p associée à $p(t)$.*

Une occurrence de Ψ-variable $\Psi_p^{[t]}$ dénote la liste des arguments t qui occurrent dans la forme littérale $p(t)$. La notation sous forme de Ψ-variable autorise la prise en compte du terme t qui constitue une restriction du domaine dans lequel la Ψ-variable Ψ_p prend ses valeurs. Elle met également en évidence la localisation des variables au niveau de la clause (cf. 6.4.1).

Définition 6.1.2 - Ψ-Domaine - *Chaque Ψ-variable Ψ_p ou $\Psi_{\neg p}$ prend ses valeurs dans le Ψ-domaine $dom(\Psi_{p,\neg p})$ qui lui est associé. Chaque Ψ-domaine est formé de deux sous-domaines $dom(\Psi_p)$ et $dom(\Psi_{\neg p})$ tels que :*

$$\begin{array}{rcll} dom(\Psi_{p,\neg p}) & = & dom(\Psi_p) \cup dom(\Psi_{\neg p}) & \\ avec & & dom(\Psi_p) \cap dom(\Psi_{\neg p}) & = \emptyset \end{array}$$

Le symbole $\cup$ ne représente pas l'union des sous-ensembles au sens mathématique, mais doit être compris comme une notation introduisant la séparation des sous-ensembles.

Exemple

$$p(X) \xrightarrow{\Psi} \Psi_p^{[X]} \qquad q(a,Y) \xrightarrow{\Psi} \Psi_q^{[(a,Y)]}$$

Définition 6.1.3 - Transformation Tr_1 - *Tout problème P du calcul des prédicats du premier ordre, exprimé sous forme normale conjonctive, composé de n_p symboles de prédicats, m clauses dont m_u clauses unitaires, est modélisable sous forme de Problème de Satisfaction de Contraintes $CSP^T\mathcal{P} = \langle \mathcal{V},\mathcal{D},\mathcal{C} \rangle$, à valeurs dans l'univers des termes T de P, par application des règles suivantes :*

- *pour chaque symbole de prédicat p de P :*
  - *$\mathcal{V}$ est composé des variables Ψ_p et $\Psi_{\neg p}$, prenant leurs valeurs dans $dom(\Psi_{p,\neg p})$, et reliées aux littéraux de la forme $p(t)$ et $\neg p(t)$.*
  - *$\mathcal{D}$ est composé des domaines $dom(\Psi_{p,\neg p})$. Chaque $dom(\Psi_{p,\neg p})$ est lui-même décomposé en deux sous-domaines $dom(\Psi_{p,\neg p}) = dom(\Psi_p) \cup dom(\Psi_{\neg p})$ tels que $dom(\Psi_p) \cap dom(\Psi_{\neg p}) = \emptyset$, définis sur T, par ;*
    - *$dom(\Psi_p) = \{t, tels\ que\ I \models p(t)\}$, représente l'ensemble des termes qui sont vrais pour une interprétation partielle I de P,*
    - *$dom(\Psi_{\neg p}) = \{t, tels\ que\ I \models \neg p(t)\}$, représente l'ensemble des termes qui sont faux pour une interprétation partielle I de P,*
  - *$\mathcal{C}$ contient initialement les contraintes de consistance de la forme $\# < 1, 1, \Psi_p, \Psi_{\neg p} >$ qui sont chargées de maintenir la consistance au niveau logique ;*
- *les clauses unitaires de P définissent initialement les domaines :*
  - *$dom(\Psi_p) = dom(\Psi_p) \cup \{t_0\}$, pour chaque clause unitaire $p(t_0)$ de P,*
  - *$dom(\Psi_{\neg p}) = dom(\Psi_{\neg p}) \cup \{t_0\}$, pour chaque clause unitaire $\neg p(t_0)$ de P,*
- *les clauses non unitaires sont transformées en contraintes de cardinalité. Une clause $p_1(t_1) \vee \ldots \vee p_n(t_n)$ est transformée en $\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >$.*

Remarque 1 - Les contraintes de consistance ne participent pas à la résolution du problème. Elles garantissent la consistance du CSP au niveau du problème logique P.

Remarque 2 - La décomposition des domaines $dom(\Psi_{p,\neg p})$ en deux sous-domaines notés $dom(\Psi_p)$ et $dom(\Psi_{\neg p})$ est déterminante car elle réalise, comme nous le verrons plus loin, un partitionnement menant à une interprétation des Ψ-variables instanciées.

Définition 6.1.4 - Terminologie - *Dans la suite de ce chapitre, bien que les domaines soient composés de termes logiques, nous emploierons indifféremment les mots* valeur *ou* terme *pour désigner un élément d'un domaine.*

Proposition 6.1.1 - *Tr_1 est une transformation linéaire.*

Preuve Si l'on considère qu'un problème sous forme normale conjonctive P est composé de n_p prédicats et m clauses dont m_u clauses unitaires, la réécriture du problème initial en CSP^T nécessite la création de $2n_p$ Ψ-variables de $\mathcal{V}$, de n_p Ψ-domaines de $\mathcal{D}$, ainsi que de n_p contraintes de consistance. Les domaines de $\mathcal{D}$ sont initialement constitués de m_u valeurs, les $m - m_u$ clauses restantes sont modélisées sous forme de contraintes de cardinalité.

Proposition 6.1.2 - Équivalence de modélisation entre problème logique et CSP^T - *Les représentations d'un problème en logique des prédicats du premier ordre sous forme normale conjonctive et sous forme de CSP^T sont équivalentes.*

Preuve La modélisation sous forme de CSP^T d'un problème en logique des prédicats du premier ordre, exprimé sous forme CNF est donnée par la transformation Tr_1. La transformation d'un CSP^T en problème sous forme CNF s'obtient en retranscrivant les contraintes de cardinalité en clauses, ainsi qu'en transformant les valeurs de chaque Ψ-domaine en clause unitaire.

Exemple : Soit le problème suivant :

$$P_0 = \begin{cases} p(a) \\ q(b) \\ \neg p(c) \\ \neg p(X) \vee \neg q(Y) \vee r(X,Y) \\ \neg r(c,Z) \end{cases}$$

Nous allons détailler l'application de Tr_1 sur ce problème. P_0 est composé de 3 symboles de prédicats p,q,r et de 5 clauses dont 4 clauses unitaires.

L'ensemble $\mathcal{V}$ est donc formé de $2 \times 3 = 6$ Ψ-variables $\{\Psi_p,\Psi_{\neg p},\Psi_q,\Psi_{\neg q},\Psi_r,\Psi_{\neg r}\}$. On associe un Ψ-domaine à chacune de ces variables, soit :

$$\begin{array}{lll} \Psi_p \text{ et } \Psi_{\neg p} & \text{prennent leurs valeurs dans} & dom(\Psi_{p,\neg p}) = dom(\Psi_p) \cup dom(\Psi_{\neg p}) \\ \Psi_q \text{ et } \Psi_{\neg q} & \text{prennent leurs valeurs dans} & dom(\Psi_{q,\neg q}) = dom(\Psi_q) \cup dom(\Psi_{\neg q}) \\ \Psi_r \text{ et } \Psi_{\neg r} & \text{prennent leurs valeurs dans} & dom(\Psi_{r,\neg r}) = dom(\Psi_r) \cup dom(\Psi_{\neg r}) \end{array}$$

Les clauses unitaires définissent initialement les sous-domaines comme suit :

$$\begin{array}{lrl} \text{pour} & p(a), & dom(\Psi_p) = \{a\} \\ \text{pour} & q(b), & dom(\Psi_q) = \{b\} \\ \text{pour} & \neg p(c), & dom(\Psi_{\neg p}) = \{c\} \\ \text{pour} & \neg r(c,Z), & dom(\Psi_{\neg r}) = \{(c,Z)\} \end{array}$$

Pour chaque symbole de prédicat, on crée une contrainte de consistance :

$$\begin{array}{ll} \text{pour}\, p & \#<1,1,\Psi_p,\Psi_{\neg p}> \\ \text{pour}\, q & \#<1,1,\Psi_q,\Psi_{\neg q}> \\ \text{pour}\, r & \#<1,1,\Psi_r,\Psi_{\neg r}> \end{array}$$

Enfin, l'unique clause non unitaire du problème est transformée en clause de cardinalité :

$$\#<1,3,\Psi_{\neg p}^{[X]},\Psi_{\neg q}^{[Y]},\Psi_r^{[X,Y]}>$$

Par application de Tr_1, P_0 se modélise donc en :

$$CSP^T\mathcal{P}_0 \left\{ \begin{array}{l} \mathcal{V} = \{\Psi_p,\Psi_{\neg p},\Psi_q,\Psi_{\neg q},\Psi_r,\Psi_{\neg r}\} \\ \mathcal{D} = \left\{ \begin{array}{l} dom(\Psi_{p,\neg p}) = \{a\} \cup \{c\} \\ dom(\Psi_{q,\neg q}) = \{b\} \cup \{\} \\ dom(\Psi_{r,\neg r}) = \{\} \cup \{(c,Z)\} \end{array} \right\} \\ \mathcal{C} = \left\{ \begin{array}{l} \left. \begin{array}{l} \#<1,1,\Psi_p,\Psi_{\neg p}> \\ \#<1,1,\Psi_q,\Psi_{\neg q}> \\ \#<1,1,\Psi_r,\Psi_{\neg r}> \end{array} \right\} \text{contraintes de consistance} \\ \#<1,3,\Psi_{\neg p}^{[X]},\Psi_{\neg q}^{[Y]},\Psi_r^{[X,Y]}> \end{array} \right\} \end{array} \right.$$

À partir de l'expression du $CSP^T\mathcal{P}_0$, on retrouve aisément l'expression du problème initial P_0, les contraintes de consistance n'étant pas retraduites.

6.2 Satisfaction de contraintes dans le cadre classique

L'idée générale à la base de *Sacre* est la modélisation et la résolution des problèmes du calcul des prédicats par des techniques CSP. En particulier cela nécessite de traduire le *principe de résolution* dans ce cadre de travail. La *résolution binaire* génère une nouvelle clause, appelée résolvante, par élimination de littéraux complémentaires unifiables de deux clauses parentes.

$$\frac{A \vee B \quad \neg A \vee C}{B \vee C}$$

L'inconsistance d'un problème est prouvée par la génération de la clause vide. Mais pour générer la clause vide, on doit finalement inférer deux clauses **unitaires** complémentaires unifiables (par exemple $\neg p(a,X)$ et $p(Y,f(Z))$. Cette dernière propriété, se traduit au niveau CSP, par l'impossibilité pour une valeur d'appartenir simultanément à des domaines complémentaires, par exemple $dom(\Psi_p)$ et $dom(\Psi_{\neg p})$, associés au prédicat p et liés respectivement aux littéraux positifs et négatifs de p.

6.2.1 Définitions

Les domaines des variables des CSP^T faisant partie de l'univers des termes logiques d'un problème, il convient d'exprimer l'ensemble des notions CSP dans le cadre nouvellement défini. Nous introduisons dans ce but quelques notations préalables :

Définition 6.2.1 - Instanciation - *Instancier une occurrence de Ψ-variable $\Psi_p^{[t]}$ par une valeur (ou un terme) μ de $dom(\Psi_{p,\neg p})$, c'est trouver un plus grand unificateur σ tel que $\sigma(t) = \sigma(\mu)$. L'instanciation de $\Psi_p^{[t]}$ par μ est alors notée :*

$$\Psi_p^{[t]} = \mu \qquad \textit{ou bien} \qquad \Psi_p^{[\sigma(t)]}$$

Définition 6.2.2 - *En conséquence, pour un littéral p, on note :*

- $U_v(\Psi_p^{[t]}) = \{(\sigma,\mu)$, *tels que* σ *est un mgu de* t,μ, *où* $\mu \in dom(\Psi_p)\}$, *l'ensemble des couples unificateur/valeur qui instancient* $\Psi_p^{[t]}$ *à une valeur* μ *de* $dom(\Psi_p)$,
- $U_f(\Psi_p^{[t]}) = \{(\sigma,\mu)$, *tels que* σ *est un mgu de* t,μ, *où* $\mu \in dom(\Psi_{\bar{p}})\}$, *l'ensemble des couples unificateur/valeur qui instancient* $\Psi_p^{[t]}$ *à une valeur* μ *de* $dom(\Psi_{\bar{p}})$ *($\bar{p}$ représente ici le complémentaire de p),*

- $U(\Psi_p^{[t]}) = \{(\sigma,\mu),$ *tels que* σ *est un mgu de* t,μ, *où* $\mu \in dom(\Psi_{p,\neg p})\} = U_v(\Psi_p^{[t]}) \cup U_f(\Psi_p^{[t]})$.

Définition 6.2.3 - Interprétation d'une variable instanciée - *L'interprétation d'une occurrence de* Ψ*-variable* $\Psi_p^{[t]}$ *par une valeur* (σ,μ) *de* $U(\Psi_p^{[t]})$ *est définie par :*

$$|\Psi_p^{[\sigma(t)]}| = \begin{cases} 1\,(ou\,vrai), & si \quad (\sigma,\mu) \in U_v(\Psi_p^{[t]}) \\ 0\,(ou\,faux), & si \quad (\sigma,\mu) \in U_f(\Psi_p^{[t]}) \end{cases}$$

Une Ψ*-variable instanciée est dite vérifiée si elle est interprée à vrai. Elle est dite falsifiée si elle est interprétée à faux.*

Par extension on notera $|\# < 1, n, \Psi_{p_1}^{[\sigma(t_1)]}, \ldots, \Psi_{p_n}^{[\sigma(t_n)]} > |$, *le nombre de* Ψ*-variables instanciées et interprétées à* vrai *par le mgu* σ *por la contrainte* $\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >$.

Définition 6.2.4 - Consistance d'un CSP^T **-** *Un* CSP^T *est dit consistant (ou satisfiable) si aucune des contraintes de consistance n'est violée, c'est-à-dire si sa résolution ne permet pas d'établir :*

$$\exists p, \exists(\sigma,\mu) \in U_v(\Psi_p^{[t]})\; et\, (\omega,\nu) \in U_f(\Psi_p^{[t]}) \quad tels\ que \quad \Psi_p^{[\sigma(t)]} = \Psi_p^{[\omega(t)]} \tag{6.1}$$

Le CSP^T *est dit inconsistant dans le cas contraire.*

L'expression 6.1 représente la traduction de l'inconsistance du problème logique initial dans le cadre des CSP^T.

Définition 6.2.5 - Satisfaction de Contrainte - *Une contrainte* $\# < \alpha, \beta, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >$ *est dite satisfaite si étant donnée une permutation* $\{i_1, \ldots, i_n\}$ *de* $\{1, \ldots, n\}$, *on a :*

$$\begin{array}{l} \exists(\sigma_{i_1},\mu_{i_1}) \in U(\Psi_{p_{i_1}}^{[t_{i_1}]}) \\ \exists(\sigma_{i_2},\mu_{i_2}) \in U(\Psi_{p_{i_2}}^{[\sigma_{i_1}(t_{i_2})]}) \\ \vdots \\ \exists(\sigma_{i_n},\mu_{i_n}) \in U(\Psi_{p_{i_n}}^{[\sigma_{i_1} \circ \sigma_{i_2} \circ \ldots \circ \sigma_{i_{n-1}}(t_{i_n})]}) \end{array}$$

tels que, pour $\sigma = \sigma_{i_1} \circ \sigma_{i_2} \circ \ldots \circ \sigma_{i_n}$, *on ait :*

$$\alpha \leq |\# < \alpha, \beta, \Psi_{p_1}^{[\sigma(t_1)]}, \ldots, \Psi_{p_n}^{[\sigma(t_n)]} > | \leq \beta$$

On note alors $\Sigma(\# < \alpha, \beta, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >) = \{\sigma, \sigma = \sigma_{i_1} \circ \sigma_{i_2} \circ \ldots \circ \sigma_{i_n}\}$, *l'ensemble des plus grands unificateurs instanciant chacune des variables et satisfaisant la contrainte.*

En d'autres termes, il existe une instanciation des variables de la contrainte telle qu'au moins α et au plus β variables soient interprétées à vrai.

Exemple : Soit le CSP^T suivant :

$$CSP^T\mathcal{P}_1 \begin{cases} \mathcal{V} = & \{\Psi_p, \Psi_{\neg p}, \Psi_q, \Psi_{\neg q}, \Psi_r, \Psi_{\neg r}\} \\ \mathcal{D} = & \left\{ \begin{array}{lccc} dom(\Psi_{p,\neg p}) = & \{a,c\} & \cup & \{b\} \\ dom(\Psi_{q,\neg q}) = & \{a,c\} & \cup & \{b\} \\ dom(\Psi_{r,\neg r}) = & \{a\} & \cup & \{b,c\} \end{array} \right\} \\ \mathcal{C} = & \left\{ \begin{array}{l} \# < 1, 1, \Psi_p, \Psi_{\neg p} > \\ \# < 1, 1, \Psi_q, \Psi_{\neg q} > \\ \# < 1, 1, \Psi_r, \Psi_{\neg r} > \\ \# < 1, 3, \Psi_{\neg p}^{[X]}, \Psi_{\neg q}^{[X]}, \Psi_r^{[X]} > \end{array} \right\} \end{cases}$$

Il existe pour ce CSP trois instanciations possibles, dont deux seulement satisfont la contrainte $\# < 1, 3, \Psi_{\neg p}^{[X]}, \Psi_{\neg q}^{[X]}, \Psi_r^{[X]} >$:

- cas $X = a$:

$$I_1 \left| \begin{array}{l} (\sigma_1 = \{a/X\},\mu_1 = a) \in U_f(\Psi_{\neg p}^{[X]}), |\Psi_{\neg p}^{[a]}| = 0, \\ (\sigma_2 = \{a/X\},\mu_2 = a) \in U_f(\Psi_{\neg q}^{[X]}), |\Psi_{\neg q}^{[a]}| = 0, \\ (\sigma_3 = \{a/X\},\mu_3 = a) \in U_v(\Psi_{r}^{[X]}), |\Psi_{r}^{[a]}| = 1, \\ |\#< 1,3,\Psi_{\neg p}^{[a]},\Psi_{\neg q}^{[a]},\Psi_{r}^{[a]} > | = 1 \in [1,3] \end{array} \right.$$

- cas $X = b$:

$$I_2 \left| \begin{array}{l} (\sigma_1 = \{b/X\},\mu_1 = b) \in U_v(\Psi_{\neg p}^{[X]}), |\Psi_{\neg p}^{[b]}| = 1, \\ (\sigma_2 = \{b/X\},\mu_2 = b) \in U_v(\Psi_{\neg q}^{[X]}), |\Psi_{\neg q}^{[b]}| = 1, \\ (\sigma_3 = \{b/X\},\mu_3 = b) \in U_f(\Psi_{r}^{[X]}), |\Psi_{r}^{[b]}| = 0, \\ |\#< 1,3,\Psi_{\neg p}^{[b]},\Psi_{\neg q}^{[b]},\Psi_{r}^{[b]} > | = 2 \in [1,3] \end{array} \right.$$

- cas $X = c$:

$$I_3 \left| \begin{array}{l} (\sigma_1 = \{c/X\},\mu_1 = c) \in U_f(\Psi_{\neg p}^{[X]}), |\Psi_{\neg p}^{[c]}| = 0, \\ (\sigma_2 = \{c/X\},\mu_2 = c) \in U_f(\Psi_{\neg q}^{[X]}), |\Psi_{\neg q}^{[c]}| = 0, \\ (\sigma_3 = \{c/X\},\mu_3 = c) \in U_f(\Psi_{r}^{[X]}), |\Psi_{r}^{[c]}| = 0, \\ |\#< 1,3,\Psi_{\neg p}^{[c]},\Psi_{\neg q}^{[c]},\Psi_{r}^{[c]} > | = 0 \notin [1,3] \end{array} \right.$$

6.3 Corrélation entre problème logique et CSP^T

Muni du formalisme précédent, notre principal objectif consiste alors à établir une correspondance entre la consistance d'un CSP^T et la consistance du problème logique qui lui correspond.

6.3.1 Inadéquation de la satisfaction de contrainte dans le cadre classique CSP

Considérons le problème **insatisfiable** suivant :

$$(P_2) \left\{ \begin{array}{l} p(a) \\ p(f(a)) \\ \neg p(X) \vee p(f(X)) \\ \neg p(f(f(f(a)))) \end{array} \right.$$

ce problème est transfomé par Tr_1 en :

$$CSP^T\mathcal{P}_2 \left\{ \begin{array}{ll} \mathcal{V} = & \{\Psi_p,\Psi_{\neg p}\} \\ \mathcal{D} = & \{dom(\Psi_{p,\neg p}) = \{a,f(a)\} \cup \{f(f(f(a)))\}\} \\ \mathcal{C} = & \left\{ \begin{array}{l} \#< 1,1,\Psi_p,\Psi_{\neg p} > \\ \#< 1,2,\Psi_{\neg p}^{[X]},\Psi_p^{[f(X)]} >\} \end{array} \right\} \end{array} \right.$$

Dans le cadre classique de la résolution des CSP, un problème est satisfiable si toutes ses contraintes sont satisfaites. Le problème $\mathcal{P}_2$ est composé d'une seule contrainte dont la seule instanciation possible est :

$$I \left| \begin{array}{l} (\sigma_1 = \{a/X\},\mu_1 = a) \in U_f(\Psi_{\neg p}^{[X]}), |\Psi_{\neg p}^{[a]}| = 0, \\ (\sigma_2 = \{a/X\},\mu_2 = a) \in U_v(\Psi_{p}^{[X]}), |\Psi_{p}^{[f(a)]}| = 1, \\ |\#< 1,2,\Psi_{\neg p}^{[a]},\Psi_{p}^{[f(a)]} > | = 1 \in [1,2] \end{array} \right.$$

Ce problème est donc satisfiable pour une approche classique de résolution des CSP^T. En outre, si l'on supprime la clause $p(f(a))$ dans le problème sous forme CNF précédent, le CSP^T correspondant ne peut être résolu car on se trouve dans l'impossibilité d'instancier l'une des Ψ-variables de la contrainte $\# < 1,2,\Psi_{\neg p}^{[X]},\Psi_p^{[f(X)]} >$, en l'occurrence, la variable $\Psi_p^{[f(X)]}$. Il apparaît donc nécessaire de modifier le cadre classique pour pouvoir résoudre spécifiquement les CSP^T.

6.4 Nouveau cadre de résolution des CSP^T

Le cadre classique de résolution des CSP n'étant pas adapté à la résolution des problèmes du calcul des prédicats exprimés sous forme de CSP^T, nous examinons dans cette section les raisons de son inadéquation.

6.4.1 Raisons de l'inadéquation du cadre classique

On peut semble-t-il distinguer quatre critères qui font la spécificité des CSP^T :

- **domaine du discours :**
  Le domaine du discours des CSP^T est l'univers des termes logique du problème. Ce domaine est potentiellement infini dans le cas où le problème comporte des termes fonctionnels, alors que dans le cadre classique les domaines sont finis.
- **portée des variables :**
  Contrairement aux CSP du calcul propositionnel et aux CSP en général, les variables sont non globales. En ce sens que si on attribue une valeur à une variable, cette instanciation ne se propage pas sur l'ensemble des contraintes du problème. Cette propriété est due au caractère particulier des Ψ-variables. Afin d'illustrer notre propos, considérons par exemple les deux contraintes suivantes :

  $$\#< 1, 2, \Psi_p^{[f(X),Y]}, \Psi_q^{[X]} >$$
  $$\#< 1, 2, \Psi_p^{[g(X),h(X)]}, \Psi_r^{[X]} >$$

  Si on instancie $\Psi_p^{[f(X),Y]}$ dans la première contrainte par la valeur $[f(a),b]$, instancier $\Psi_p^{[g(X),h(X)]}$ dans la deuxième contrainte perd tout son sens puisque les deux premiers termes $f(X)$ et $g(X)$ ne sont pas unifiables. Dans le cadre classique des CSP, une variable instanciée garde la même valeur dans toutes les contraintes où elle apparaît. Ce n'est pas le cas ici.
- **déduction, propagation :**
  En calcul propositionnel, il existe deux cas de déduction des clauses de cardinalité (on parlera de propagation dans le cas des CSP). Soit la contrainte de cardinalité :

  $$\#< \alpha, \beta, p_1, \ldots, p_n >$$

  On pourra au cours de la résolution de cette contrainte se retrouver dans l'une des situations suivantes :

  a) soit β littéraux de la clause de cardinalité sont instanciés à *vrai*, on en déduit que pour que la contrainte soit satisfaite, il faut nécessairement que les $n - \beta$ (où n est la longueur de la contrainte) littéraux non encore instanciés, soient instanciés à *faux*.
  b) soit $n - \alpha$ littéraux de la clause de cardinalité sont instanciés à *faux*, la contrainte est alors satisfaite si les α littéraux restants sont instanciés à *vrai*.

  En CSP^T, les clauses de cardinalité utilisées pour résoudre le problème sont des clauses (1,n). Les clauses de consistance, comme leur nom l'indique, qui sont des clauses (1,1) ne sont utilisées que pour vérifier la consistance du problème. Une contrainte (1,n) est satisfaite si au moins une de ses variables est instanciée par une valeur interprétant la variable à vrai. Les variables restantes peuvent prendre n'importe quelle valeur ne violant pas les autres contraintes. Dans le cas des CSP^T, l'incidence de la portée des variables a pour conséquence que, dès qu'une variable est instanciée à *vrai*, il est inutile de continuer à instancier à *vrai* les aux autres variables de la contrainte puisque celle-ci est trivialement satisfaite et qu'il n'y a aucune répercussion sur les autres contraintes. Il existe cependant un cas de figure où la satisfaction de contrainte se révèle intéressante et qui correspond à la situation de propagation **b)** décrite précédemment : supposons que $n - 1$ littéraux de la contrainte aient été instanciés à $faux$. Pour que la contrainte soit satisfaite, il est nécessaire que le dernier littéral à instancier prenne la valeur $vrai$. Ce cas de propagation fournit donc un procédé par lequel on pourra explicitement établir la valeur de vérité de certains littéraux.

- **extension des domaines :**
L'extension des domaines découle directement des trois critères précédents, ainsi pour le problème :

$$CSP^T\mathcal{P}_3 \left\{ \begin{array}{lll} \mathcal{V} & = & \{\Psi_p,\Psi_{\neg p},\Psi_q,\Psi_{\neg q},\Psi_r,\Psi_{\neg r}\} \\ \mathcal{D} & = & \left\{ \begin{array}{lcccc} dom(\Psi_{p,\neg p}) = & \{a\} & \cup & \{\} \\ dom(\Psi_{q,\neg q}) = & \{a\} & \cup & \{\} \\ dom(\Psi_{r,\neg r}) = & \{\} & \cup & \{\} \end{array} \right\} \\ \mathcal{C} & = & \left\{ \begin{array}{l} \#< 1,1,\Psi_p,\Psi_{\neg p} > \\ \#< 1,1,\Psi_q,\Psi_{\neg q} > \\ \#< 1,1,\Psi_r,\Psi_{\neg r} > \\ \#< 1,3,\Psi_{\neg p}^{[X]},\Psi_{\neg q}^{[X]},\Psi_r^{[f(X)]} > \end{array} \right\} \end{array} \right.$$

l'instanciation partielle de l'unique contrainte du problème :

$$I \left| \begin{array}{l} \sigma_1 = \{a/X\} \in U_f(\Psi_{\neg p}^{[X]}), |\Psi_{\neg p}^{[a]}| = 0, \\ \sigma_2 = \{a/X\} \in U_f(\Psi_{\neg q}^{[X]}), |\Psi_{\neg q}^{[a]}| = 0, \end{array} \right.$$

sera satisfaite si on pose $|\Psi_r^{[f(a)]}| = 1$, soit $dom(\Psi_r) = \{f(a)\}$. Or $f(a)$ n'appartient pas à $dom(\Psi_r)$. Nous sommes donc contraints d'ajouter une nouvelle valeur au domaine $dom(\Psi_r)$ pour pouvoir résoudre le CSP^T.

6.4.2 CSP^T étendus

L'ensemble des raisons que nous venons d'invoquer, démontrent que la résolution des CSP^T est spécifique et requiert une adaptation des procédés de résolution des CSP^T. En l'occurrence, la résolution des contraintes par propagation qui correspond au cas de propagation **b)**, vu dans la section précédente, fournit un procédé univoque de satisfaction des contraintes pour lequel il est nécessaire d'étendre les domaines.

Définition 6.4.1 - CSP^T à domaines étendus - *Un CSP^T à domaines étendus (noté CSP^T_{ext}) est un CSP^T dont les domaines sont susceptibles d'être modifiés par l'ajout de nouvelles valeurs au cours de la résolution par la satisfaction des contraintes.*

Un CSP^T et un CSP^T_{ext} ont donc la même représentation, mais diffèrent suivant la méthode de résolution employée.

On peut également appliquer la notion de CSP étendu à d'autres classes de CSP, comme les CSP sur le domaine entier. Considérons le problème en domaine fini entier suivant :

$$CSP \left\{ \begin{array}{l} \mathcal{V} = \{x_1,x_2,x_3\} \\ \mathcal{D} = \{d_1,d_2,d_3\} \text{ tels que } d_1 = d_2 = d_3 = \{0,1,2\} \\ \mathcal{C} = \{x_1 + x_2 = x_3\} \end{array} \right.$$

les solutions de ce CSP sont :

$$S = \{(0,0,0),(0,1,1),(1,0,1),(1,1,2),(0,2,2),(2,0,2)\}$$

Dans le cadre classique des CSP, l'instanciation $x_1 = 1$ et $x_2 = 2$ est inconsistante car $x_1 + x_2 = 3$ n'appartient pas au domaine d_3 de x_3. En revanche, si l'on suit une approche étendue, l'instanciation précédente est valide et la valeur 3 est ajoutée au domaine d_3. Les solutions, après une première itération de la résolution étendue, sont :

$$S_{ext} = S \cup \{(1,2,3),(2,1,3),(2,2,4)\}$$

En itérant ce procédé d'extension, les domaines d_1,d_2,d_3 seront étendus en $d_1,d_2,d_3 = N$.

Théorème 6.4.1 - Correction des CSP^T_{ext} - *Soit P un problème de la logique des prédicats du premier ordre, et $\mathcal{P}$, le CSP^T_{ext} correspondant à P. Si $\mathcal{P}$ est inconsistant, alors P est inconsistant.*

Preuve Si $\mathcal{P}$ est inconsistant, cela signifie qu'une contrainte de consistance est violée, soit, d'après la définition 6.2.4 qu'il existe un terme $\mu \in dom(\Psi_{p,\neg p})$, pour un prédicat donné p, tel que μ appartienne à la fois à $dom(\Psi_p)$ et $dom(\Psi_{\neg p})$ à une substitution près. En d'autres termes, et conformément à la définition 6.1.3, il existe deux clauses unitaires $p(t_1)$ et $\neg p(t_2)$ telles que t_1 et t_2 soient unifiables, ce qui selon le *principe de résolution*, prouve l'inconsistance du problème P.

6.5 Résolution des CSP^T_{ext}

Les critères d'inadéquation suggèrent que la résolution des CSP^T_{ext} se conçoive selon la propagation des valeurs issues de la satisfaction des contraintes suivant le cas de propagation **b)**.

Le coeur de la méthode de résolution des CSP^T_{ext} consiste schématiquement à déterminer par propagation la valeur de vérité des valeurs qui étendent les domaines.

On procède en sélectionnant une Ψ-variable dont le domaine associé est non vide et on résout toutes les contraintes où cette variable occure. On peut également choisir, de manière similaire, de résoudre l'ensemble des contraintes où le complémentaire ce cette variable apparaît.

Résoudre une contrainte pour une variable donnée consiste à falsifier l'occurrence de la Ψ-variable associée à la variable en question et à satisfaire la contrainte résultante de manière à ce qu'une seule occurrence de Ψ-variable soit vérifiée, comme explicité précédemment.

Chaque nouvelle valeur qui vérifie une variable lors de la satisfaction de la contrainte vient augmenter le domaine associé à cette variable. Elle sera ensuite réutilisée lors de futures propagations entraînant une sorte de *réaction en chaîne*.

Comme on peut le constater, la méthode de résolution choisie est axée sur les domaines. On peut envisager d'axer la résolution des CSP^T_{ext} sur les contraintes. Néanmoins, le système qui en découle est assez complexe et difficilement formalisable. Nous donnons en annexe un aperçu du fonctionnement d'un tel système. En outre, son implantation entraîne une complexité spatiale nettement plus importante que le système que nous décrivons dans cette section.

6.5.1 Sémantique opérationnelle

Nous définissons à présent la sémantique opérationnelle des CSP^T à domaines étendus.

Définition 6.5.1 - Partition des domaines - *Chacun des sous-domaines* $dom(\Psi_p)$ *(resp.* $dom(\Psi_{\neg p})$*) de* $\mathcal{D}$ *est partitionné en* $dom^T(\Psi_p) \cup dom^N(\Psi_p)$ *(resp.* $dom^T(\Psi_{\neg p}) \cup dom^N(\Psi_{\neg p})$*).*

- $dom^N(\Psi_p)$ *représente les valeurs/termes non traitées de* $dom(\Psi_p)$,
- $dom^T(\Psi_p)$ *représente les valeurs/termes traitées de* $dom(\Psi_p)$,

Définition 6.5.2 - Etat du système - *On définit l'état du système par un couple* $(V,\mathcal{D})$ *où*

- V *est un ensemble de variables à traiter.*
- $\mathcal{D}$ *est un ensemble de domaines, comme suit :*

$$\mathcal{D} = \left\{ \begin{array}{l} dom(\Psi_{p_1,\neg p_1}) = dom(\Psi_{p_1}) \cup dom(\Psi_{\neg p_1}) \\ \ldots \\ dom(\Psi_{p_n,\neg p_n}) = dom(\Psi_{p_n}) \cup dom(\Psi_{\neg p_n}) \end{array} \right\}$$

sachant que chaque $dom(\Psi_{p_i})$ *(et* $dom(\Psi_{\neg p_i})$*) est décomposé en* $dom^N(\Psi_{p_i}) \cup dom^T(\Psi_{p_i})$,

Définition 6.5.3 - Etat initial du système - *L'état initial du système est donné par :*

- $\mathcal{D}$ *est défini suivant la définition 6.1.3, de manière à ce que* $dom^N(\Psi_{p_i}) = dom(\Psi_{p_i})$, $dom^T(\Psi_{p_i}) = \emptyset$ *et* $dom^N(\Psi_{\neg p_i}) = dom(\Psi_{\neg p_i})$, $dom^T(\Psi_{\neg p_i}) = \emptyset Moi$ *pour tout* $1 \leq i \leq n$.
- V *est formé des variables* Ψ_{p_i} *telles que* $dom^N(\Psi_{p_i}) \neq \emptyset$ *ou* $dom^N(\Psi_{\neg p_i}) \neq \emptyset$.

Définition 6.5.4 - Règle de propagation - *La règle de propagation définit une transformation d'un état initial en un état final. En d'autres termes, elle réalise la propagation des valeurs issues de déductions précédentes.*

$$\frac{\begin{array}{l} \qquad\qquad\qquad \exists \Psi_{p_v} \in V \textit{ telle que}, dom^N(\Psi_{p_v}) \neq \emptyset \\ \qquad\qquad C(\Psi_{p_v}) = \{\#< 1, n, \Psi p_{i_1} t_{i_1}, \ldots, \Psi_{p_{i_{n-1}}}^{[t_{i_{n-1}}]}, \Psi_{\neg p_v}^{[t_v]} >\} \neq \emptyset \\ \forall c \in C(\Psi_{p_v}), \\ \forall k \in [1, n-1], \\ dom'^N(\Psi_{p_{i_k}}) = dom^N(\Psi_{p_{i_k}}) \cup \{\sigma(t_{i_k}), \quad \forall \sigma \in \Sigma(\#< 1, 1, \Psi_{p_{i_1}}^{[\omega(t_{i_1})]}, \ldots, \Psi_{p_{i_l}}^{[\omega(t_{i_l})]}, \Psi_{\neg p_v}^{[\omega(t_v)]} >) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall (\omega, \mu) \in U_f(\Psi_{\neg p_v}^{[t_v]}), \textit{tel que} \quad \Psi_{\neg p_v}^{[t_v]} = \mu\} \\ dom'^T(\Psi_{p_v}) = dom^T(\Psi_{p_v}) \cup dom^N(\Psi_{p_v}) \\ dom'^N(\Psi_{p_v}) = \emptyset \\ V' = V \cup \{\Psi_{p_i}, dom'^N(\Psi_{p_i}) \neq \emptyset\} \end{array}}{(V, \mathcal{D}) \quad \longrightarrow \quad (V', \mathcal{D}')} \ (Prop)$$

Définition 6.5.5 - Règle d'inconsistance - *La règle d'inconsistance signale toute inconsistance sur les domaines.*

$$\frac{\exists p, \exists \mu \in dom(\Psi_p) \quad \exists \nu \in dom(\Psi_{\neg p}) \quad \exists \sigma, \omega \textit{ tels que } \sigma(\mu) = \omega(\nu)}{(V, D) \longrightarrow \perp} \ (Inc)$$

6.5.2 Exemple

Soit le problème logique suivant :

$$P_5 \left\{ \begin{array}{l} p(a) \\ p(b) \\ q(f(a)) \\ \neg p(X) \vee \neg q(f(X)) \vee r(X) \\ \neg p(X) \vee s(X) \\ r(f(b)) \\ \neg r(X) \vee \neg s(Y) \vee t(X, Y) \\ \neg t(f(X), X) \end{array} \right.$$

Le CSP_{ext}^T associé à ce problème est :

$$CSP^T \mathcal{P}_4 \left\{ \begin{array}{lcl} \mathcal{X} & = & \{\Psi_p, \Psi_{\neg p}, \Psi_q, \Psi_{\neg q}, \Psi_r, \Psi_{\neg r}, \Psi_s, \Psi_{\neg s}, \Psi_t, \Psi_{\neg t}, \} \\ \\ \mathcal{D} & = & \left\{ \begin{array}{llcl} dom(\Psi_{p,\neg p}) & = \{a, b\} & \cup & \{\} \\ dom(\Psi_{q,\neg q}) & = \{f(a)\} & \cup & \{\} \\ dom(\Psi_{r,\neg r}) & = \{f(b)\} & \cup & \{\} \\ dom(\Psi_{s,\neg s}) & = \{\} & \cup & \{\} \\ dom(\Psi_{t,\neg t}) & = \{\} & \cup & \{(f(X), X)\} \end{array} \right\} \\ \\ \mathcal{C} & = & \left\{ \begin{array}{ll} & \#< 1, 1, \Psi_p, \Psi_{\neg p} > \\ & \#< 1, 1, \Psi_q, \Psi_{\neg q} > \\ & \#< 1, 1, \Psi_r, \Psi_{\neg r} > \\ & \#< 1, 1, \Psi_s, \Psi_{\neg s} > \\ & \#< 1, 1, \Psi_t, \Psi_{\neg t} > \\ (c_1) & \#< 1, 3, \Psi_{\neg p}^{[X]}, \Psi_{\neg q}^{[f(X)]}, \Psi_r^{[X]} > \\ (c_2) & \#< 1, 2, \Psi_{\neg p}^{[X]}, \Psi_s^{[X]} > \\ (c_3) & \#< 1, 3, \Psi_{\neg r}^{[X]}, \Psi_{\neg s}^{[Y]}, \Psi_t^{[(X,Y)]} > \end{array} \right\} \end{array} \right.$$

L'état initial du système est défini par :

$$V = \{\Psi_p, \Psi_q, \Psi_r, \Psi_{\neg t}\}$$
$$\mathcal{D} = \left\{ \begin{array}{llll} dom^N(\Psi_p) = \{a,b\} & dom^T(\Psi_p) = \{\} & dom^N(\Psi_{\neg p}) = \{\} & dom^T(\Psi_{\neg p}) = \{\} \\ dom^N(\Psi_q) = \{f(a)\} & dom^T(\Psi_q) = \{\} & dom^N(\Psi_{\neg q}) = \{\} & dom^T(\Psi_{\neg q}) = \{\} \\ dom^N(\Psi_r) = \{f(b)\} & dom^T(\Psi_r) = \{\} & dom^N(\Psi_{\neg r}) = \{\} & dom^T(\Psi_{\neg r}) = \{\} \\ dom^N(\Psi_s) = \{\} & dom^T(\Psi_s) = \{\} & dom^N(\Psi_{\neg s}) = \{\} & dom^T(\Psi_{\neg s}) = \{\} \\ dom^N(\Psi_t) = \{\} & dom^T(\Psi_t) = \{\} & dom^N(\Psi_{\neg t}) = \{(f(X),X)\} & dom^T(\Psi_{\neg t}) = \{\} \end{array} \right\}$$

La règle de propagation, appliquée pour la première fois, donne par exemple :

$$\frac{\begin{array}{c} \Psi_p \in V, dom^N(\Psi_p) \neq \emptyset \\ C(\Psi_p) = \{c_1, c_2\} \\ dom'^N(\Psi_r) = dom^N(\Psi_r) \cup \{a\} \\ dom'^N(\Psi_s) = dom^N(\Psi_s) \cup \{a,b\} \\ dom'^T(\Psi_p) = dom^T(\Psi_p) \cup \{a,b\} \\ dom'^N(\Psi_p) = \{\} \\ V' = \{\Psi_q, \Psi_r, \Psi_s, \Psi_{\neg t}\} \end{array}}{(V,\mathcal{D}) \longrightarrow (V',\mathcal{D}')} \ (Prop)$$

l'état résultant de l'application de la règle de propagation est :

$$V' = \{\Psi_q, \Psi_r, \Psi_s, \Psi_{\neg t}\}$$
$$\mathcal{D}' = \left\{ \begin{array}{llll} dom^N(\Psi_p) = \{\} & dom^T(\Psi_p) = \{a,b\} & dom^N(\Psi_{\neg p}) = \{\} & dom^T(\Psi_{\neg p}) = \{\} \\ dom^N(\Psi_q) = \{f(a)\} & dom^T(\Psi_q) = \{\} & dom^N(\Psi_{\neg q}) = \{\} & dom^T(\Psi_{\neg q}) = \{\} \\ dom^N(\Psi_r) = \{f(b),a\} & dom^T(\Psi_r) = \{\} & dom^N(\Psi_{\neg r}) = \{\} & dom^T(\Psi_{\neg r}) = \{\} \\ dom^N(\Psi_s) = \{a,b\} & dom^T(\Psi_s) = \{\} & dom^N(\Psi_{\neg s}) = \{\} & dom^T(\Psi_{\neg s}) = \{\} \\ dom^N(\Psi_t) = \{\} & dom^T(\Psi_t) = \{\} & dom^N(\Psi_{\neg t}) = \{(f(X),X)\} & dom^T(\Psi_{\neg t}) = \{\} \end{array} \right\}$$

On applique une deuxième fois la règle de propagation :

$$\frac{\begin{array}{c} \Psi_s \in V', dom^N(\Psi_s) \neq \emptyset \\ C(\Psi_s) = \{c_3\} \\ dom''^N(\Psi_t) = dom'^N(\Psi_t) \cup \{(a,a), (a,b), (f(b),a), (f(b),b)\} \\ dom''^N(\Psi_{\neg r}) = dom'^N(\Psi_{\neg r}) \cup \{f(a), f(b)\} \\ dom''^T(\Psi_s) = dom'^T(\Psi_s) \cup \{a,b\} \\ dom''^N(\Psi_s) = \{\} \end{array}}{(V',\mathcal{D}') \longrightarrow (V'',\mathcal{D}'')} \ (Prop)$$

$$V'' = \{\Psi_q, \Psi_r, \Psi_t, \Psi_{\neg t}\}$$
$$\mathcal{D}'' = \left\{ \begin{array}{llll} dom^N(\Psi_p) = \{\} & dom^T(\Psi_p) = \{a,b\} & dom^N(\Psi_{\neg p}) = \{\} & dom^T(\Psi_{\neg p}) = \{\} \\ dom^N(\Psi_q) = \{f(a)\} & dom^T(\Psi_q) = \{\} & dom^N(\Psi_{\neg q}) = \{\} & dom^T(\Psi_{\neg q}) = \{\} \\ dom^N(\Psi_r) = \{f(b),a\} & dom^T(\Psi_r) = \{\} & dom^N(\Psi_{\neg r}) = \left\{ \begin{array}{l} (f(a), \\ (f(b) \end{array} \right\} & dom^T(\Psi_{\neg r}) = \{\} \\ dom^N(\Psi_s) = \{\} & dom^T(\Psi_s) = \{a,b\} & dom^N(\Psi_{\neg s}) = \{\} & dom^T(\Psi_{\neg s}) = \{\} \\ dom^N(\Psi_t) = \left\{ \begin{array}{l} (a,a), \\ (a,b), \\ (f(b),a), \\ (f(b),b) \end{array} \right\} & dom^T(\Psi_t) = \{\} & dom^N(\Psi_{\neg t}) = \{(f(X),X)\} & dom^T(\Psi_{\neg t}) = \{\} \end{array} \right\}$$

Enfin, l'application de la règle d'inconsistance démontre l'inconsistance du problème :

$$\frac{\exists \mu = (f(b),b) \in dom''(\Psi_t) \qquad \exists \nu = (f(X),X) \in dom''(\Psi_{\neg t}) \qquad \exists \sigma = \{b/X\}, \sigma(\mu) = \sigma(\nu)}{(V'',\mathcal{D}'') \longrightarrow \bot} \ (Inc)$$

6.5.3 Discussion sur l'approche de résolution des CSP_{ext}^T

Ayant montré l'équivalence de représentation entre problèmes de la logique écrits sous forme CNF et les CSP_{ext}^T, on peut se demander si la résolution des CSP_{ext}^T possède un équivalent en logique.

D'un point de vue théorique, la réécriture des problèmes logiques en CSP_{ext}^T opère une transformation d'un problème logique initial en une base de faits ($\mathcal{F}$) constituée des clauses unitaires du problèmes et d'une base de règles[54] ($\mathcal{R}$) formée des clauses non unitaires.

En effet, toute clause non unitaire $p_1(t_1) \vee \ldots \vee p_n(t_n)$ est transformée en une clause de cardinalité $\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >$ telle que si on note $Sol(c)$, l'ensemble des solutions d'une contrainte de cardinalité c, on a :

$$Sol(\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >) = \bigcup_{i=1}^{n} Sol(\# < i, i, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >)$$

Nous avons expliqué précédemment que la satisfaction des contraintes de cardinalité portant sur des termes logiques, ne possédait de réel intérêt que si elle permettait d'établir la valeur de vérité de nouveaux faits et que la sous-contrainte $\# < 1, 1, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >$ était la seule capable de répondre à cette exigence.

En conséquence, la méthode de résolution des CSP_{ext}^T que nous avons défini est donc axée sur la satisfaction des sous-contraintes (1,1) qui représentent une notation abrégée de l'ensemble des *contrapositives* d'une clause CNF :

$$\# < 1, 1, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} > \equiv \left\{ \begin{array}{l} \neg\Psi_{p_1}^{[t_1]} \wedge \neg\Psi_{p_2}^{[t_2]} \wedge \ldots \wedge \neg\Psi_{p_{n-1}}^{[t_{n-1}]} \rightarrow \Psi_{p_n}^{[t_n]} \\ \neg\Psi_{p_1}^{[t_1]} \wedge \neg\Psi_{p_2}^{[t_2]} \wedge \ldots \wedge \neg\Psi_{p_n}^{[t_n]} \rightarrow \Psi_{p_{n-1}}^{[t_{n-1}]} \\ \vdots \\ \neg\Psi_{p_2}^{[t_2]} \wedge \neg\Psi_{p_3}^{[t_3]} \wedge \ldots \wedge \neg\Psi_{p_n}^{[t_n]} \rightarrow \Psi_{p_1}^{[t_1]} \end{array} \right\}$$

L'algorithme chargé de satisfaire les contraintes (1,1) est alors équivalent à un algorithme de **chaînage avant** appliqué à une base de faits et de règles [107]. Nous donnons, figure 6.1, le principe d'un tel algorithme pour des clauses CNF en calcul propositionnel.

```
fonction Chainage_Avant(R : Base de règles, F : Base de faits ) : Base de Faits
debut
    repeter
        si ∃ une clause p1 ∨ ... ∨ pn ∨ p de R telle que ∀i,¬pi ∈ F alors
            si p ∉ F alors ajouter p à F fsi
        fsi
    jusqu'a contradiction ou aucun nouveau fait ajouté
    retourne F
fin
```

FIG. 6.1 – *Algorithme de chaînage avant en calcul propositionnel*

Le principal intérêt du *chaînage avant* est sa simplicité et son efficacité notamment sur des règles. Il consiste à produire de nouveaux faits par application des règles ou des clauses sur la base de faits.

6.5.4 Incomplétude

Proposition 6.5.1 - Incomplétude pour la résolution - *La résolution des CSP_{ext}^T est incomplète pour la résolution.*

54. le terme règle devant être compris comme une formule bien formée de la forme $p_1 \wedge \ldots \wedge p_{n-1} \rightarrow p_n$.

Preuve L'unique règle de déduction que nous employons est la *règle de propagation* qui se comporte comme l'*unit-resulting resolution* [79], valide mais incomplète. L'*unit-resulting resolution* est une spécialisation de l'*unit resolution* qui consiste à produire une nouvelle clause unitaire à partir d'un ensemble de clauses dont une seule est une clause non unitaire (noyau) alors que les autres sont des clauses unitaires (satellites). L'*unit resolution* peut être considérée comme une méta-règle appliquant une séquence de résolutions binaires, chacune d'entre-elles impliquant une clause unitaire.

En fait le *chaînage avant* est l'*unit-resulting resolution* sont très proches [107]. Elles diffèrent simplement en fonction de l'expression des clauses sur lesquelles elles s'appliquent. L'*unit-resulting resolution* s'applique sur des clauses, alors que le *chaînage avant* s'applique sur des règles.

Exemple Considérons le problème suivant, **inconsistant** par application du principe de résolution :

$$P_5 \left\{ \begin{array}{l} p(a) \\ \neg p(X) \vee q(X) \vee r(X) \\ \neg p(X) \vee q(X) \vee \neg r(X) \\ \neg p(X) \vee \neg q(X) \vee r(X) \\ \neg p(X) \vee \neg q(X) \vee \neg r(X) \end{array} \right.$$

la transformation de ce problème en CSP^T_{ext} donne :

$$CSP\mathcal{P}_5 \left\{ \begin{array}{lll} \mathcal{X} & = & \{\Psi_p, \Psi_{\neg p}, \Psi_q, \Psi_{\neg q}, \Psi_r, \Psi_{\neg r}\} \\ \\ \mathcal{D} & = & \left\{ \begin{array}{llll} dom(\Psi_{p,\neg p}) & = \{a\} & \cup & \{\} \\ dom(\Psi_{q,\neg q}) & = \{\} & \cup & \{\} \\ dom(\Psi_{r,\neg r}) & = \{\} & \cup & \{\} \end{array} \right\} \\ \\ \mathcal{C} & = & \left\{ \begin{array}{l} \#< 1, 1, \Psi_p, \Psi_{\neg p} > \\ \#< 1, 1, \Psi_q, \Psi_{\neg q} > \\ \#< 1, 1, \Psi_r, \Psi_{\neg r} > \\ \#< 1, 3, \Psi^{[X]}_{\neg p}, \Psi^{[X]}_{q}, \Psi^{[X]}_{r} > \\ \#< 1, 3, \Psi^{[X]}_{\neg p}, \Psi^{[X]}_{q}, \Psi^{[X]}_{\neg r} > \\ \#< 1, 3, \Psi^{[X]}_{\neg p}, \Psi^{[X]}_{\neg q}, \Psi^{[X]}_{r} > \\ \#< 1, 3, \Psi^{[X]}_{\neg p}, \Psi^{[X]}_{\neg q}, \Psi^{[X]}_{\neg r} > \end{array} \right\} \end{array} \right.$$

La résolution du CSP obtenu ne permet pas de déterminer l'inconsistance car on ne peut satisfaire aucune contrainte. La raison en est que seul le domaine $dom(\Psi_p)$ est défini. Toutes les contraintes étant de longueur 3, il faut nécessairement qu'au moins deux domaines soient définis pour pouvoir satisfaire une contrainte. Bien qu'incomplète, l'*unit resolution* se révèle parfois plus efficace que d'autres règles d'inférence. Ainsi le problème CAT003-4 (cf chapitre Résultats) est résolu par Otter en 161 s en employant l'hyperrésolution, alors que l'*unit resolution* permet de résoudre le même problème en moins d'une seconde.

6.6 Remédier à l'incomplétude

Nous avons déjà souligné le fait que la résolution des CSP^T_{ext} s'apparentait à l'application d'un algorithme de *chaînage avant* qui est généralement efficace.

L'inconvénient majeur du *chaînage avant* est son incomplétude. Pour y remédier au moins deux solutions s'offrent à nous.

6.6.1 Achèvement

La première connue sous le nom d'*achèvement* est proposée par Philippe **Mathieu** [78] et peut se résumer en ces termes : "*puisque le chaînage avant ne sait pas effectuer toutes les déductions que l'on attend de lui, pourquoi ne pas les lui apprendre ? En effet, il suffit d'ajouter à la base de règles les règles qui représentent des déductions que l'algorithme ne sait pas effectuer pour que le calcul devienne complet.*"

(voir [107] p. 15). Pour *achever* une base de règles, il suffit de lui adjoindre les règles manquantes pour que le *chaînage avant* soit complet. On peut alors effectuer un achèvement *total* indépendant de la base de faits, ou *partiel* si les faits sont pris en compte, ce qui permet d'éviter l'ajout de règles qui ne seront jamais utilisées. La notion d'achèvement est basée sur le calcul des *impliqués premiers* :

Définition 6.6.1 - Impliqué premier *(selon [107]) - Soit B une base (ou ensemble) de clauses. On appelle* impliqué *toute clause C telle que $B \models C$. Un* impliqué premier *est alors un impliqué qui n'est strictement subsumé par aucun autre.*

Si en calcul propositionnel l'ensemble des impliqués premiers est toujours fini, en revanche en calcul des prédicats l'ensemble des impliqués premiers peut être infini.

Pour l'ensemble de clauses suivant :

$$\begin{array}{l} p(X) \rightarrow p(f(X)) \\ p(a) \end{array}$$

l'ensemble des impliqués premiers est

$$\{p(s(a)),p(s(s(a))),p(s(s(s(a)))),\ldots\}$$

Il faut noter que la non finitude de l'ensemble des impliqués premiers ne se cantonne pas à des ensembles récursifs de clauses mais peut également subvenir pour des ensembles de clauses ne contenant pas de symbole de fonction comme par exemple pour la clause $p(X) \wedge q(X,Y) \rightarrow p(Y)$ ([107] p. 96).

Selon Olivier ROUSSEL : "*il paraît donc impossible d'achever totalement et de manière finie certaines bases, même en se limitant à des cas datalog très simples*". Olivier ROUSSEL préconise donc le recours à plusieurs alternatives face à la non finitude de la base achevée :

- on peut se restreindre aux bases qui admettent un achèvement total équivalent fini,
- sinon, uriliser des conceptions *dégradées* de l'achèvement
  - en effectuant un achèvement partiel en limitant le nombre de faits,
  - ou en effectuant un achèvement total tout en renonçant à l'équivalence sémantique entre la base et la base achevée.

Pour résumer, le principal avantage de l'*achèvement* réside dans le fait que l'exécution du *chaînage avant* sur une base achevée est plus efficace que l'utilisation d'une méthode de production. Sa principale faiblesse concerne sa complexité en temps et en espace. En effet, dans le pire des cas, la complexité temporelle de l'*achèvement* est exponentielle. En fait, l'achèvement s'applique efficacement sur des problèmes exprimés sous forme de règles.

6.6.2 Analyse de cas

L'autre alternative pour remédier à l'incomplétude du *chaînage avant*, qui est la solution que nous proposons, est une approche de *séparation par cas.*

Nous avons vu que le système de résolution basé sur la règle de propagation était incomplet. Du point de vue de la satisfaction de contraintes, nous avons mis en évidence le fait que nous n'utilisions que la sous-contrainte (1,1) d'une contrainte (1,n) lors de la satisfaction. L'ensemble des solutions d'une contrainte de cardinalité (1,n) s'exprime en fait par :

$$Sol(\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >) = \bigcup_{i=1}^{n} Sol(\# < i, i, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_n}^{[t_n]} >)$$

La forme particulière de la méthode de résolution *Sacre* nécessite que les domaines des Ψ-variables du problème à résoudre soient non vides pour initier la recherche. Or, pour certaines classes de problèmes, il s'avère que les domaines ne sont pas assez "*complets*" pour pouvoir satisfaire certaines clauses de cardinalité, ce qui a pour effet de mener à l'incomplétude. C'est par exemple le cas du CSP $\mathcal{P}_5$ vu précédemment ou du problème suivant :

$$\begin{array}{rcl} p(X) & \vee & q(X) \\ p(X) & \vee & \neg q(X) \\ \neg p(X) & \vee & q(X) \\ \neg p(X) & \vee & \neg q(X) \end{array}$$

On peut alors se demander sous quelles conditions apparaît l'incomplétude.

Proposition 6.6.1 - Condition d'incomplétude - *La recherche peut s'avérer incomplète si au moins une clause de cardinalité n'a pu être satisfaite.*

Cette condition résulte des constatations précédentes, mais ne constitue pas une condition *nécessaire et suffisante* comme en témoigne le problème suivant :

$$\begin{array}{l} r(X) \vee p(X) \vee q(X) \\ s(a) \\ \neg s(X) \vee t(X) \\ \neg t(X) \end{array}$$

pour lequel la première clause ne pourra être satisfaite en raison d'un manque d'informations dans la spécification du problème. Le cas d'incomplétude le plus généralement rencontré correspond aux problèmes spécifiant des informations incomplètes, comme :

$$\begin{array}{ll} (c_1) & \neg couleur(X) \vee dominante(X) \\ (c_2) & \neg dominante(bleu) \\ (c_3) & couleur(rouge) \vee couleur(vert) \vee couleur(bleu) \end{array}$$

Ce problème exprime le fait que si un objet est de couleur X, alors sa couleur dominante est X (c_1). Par ailleurs, on sait qu'un objet est soit de couleur *rouge*, *verte* ou *bleue* (c_3). On veut alors savoir s'il est possible qu'un objet soit de couleur dominante *bleue* (conclusion c_2). Résoudre ce problème consiste à vérifier la consistance des informations. L'utilisation du principe de résolution ne permet pas de démontrer l'inconsistance du problème. *Sacre* le peut par contre et permettrait d'établir que le problème est inconsistant si l'objet est *bleu*.

La méthode consiste à retiré la contrainte non satisfaite de l'expression du problème et à la traiter séparément. Nous sommes alors amené à faire des hypothèses sur la valeur de vérité des littéraux de la clause non satisfaite et à en évaluer les conséquences en terme de consistance.

L'algorithme correspondant au traitement des contraintes non satisfaites est donné figure 6.2. Afin d'en améliorer la compréhension, il est écrit pour une clause. Il consiste à réaliser une séparation des différents cas de figures possibles et à tester pour chacun la consistance du problème résultant.

```
fonction Séparation_de_Cas (P : Problème,p1(t1) ∨ ... ∨ pn(tn)) : booléen
pour α de 1 a n faire
    pour chaque sous-ensemble de α littéraux pi1(ti1),...,piα(tiα)
        parmi p1(t1),...,pn(tn) faire
        P' = P ∪ {pi1(ti1)} ∧ ... ∧ {piα(tiα)}
        si Consistance( P' ) = faux alors retourner inconsistant fsi
    fpour
fpour
retourner consistant
```

FIG. 6.2 – *Algorithme pour le traitement d'une contrainte non satisfaite.*

En particulier, le problème précédent se réduit à :

$$\begin{array}{ll} (c_1) & \neg couleur(X) \vee dominante(X) \\ (c_2) & \neg dominante(bleu) \end{array}$$

La contrainte $\# < 1, 3, \Psi^{[rouge]}_{couleur}, \Psi^{[vert]}_{couleur}, \Psi^{[bleu]}_{couleur} >$ qui n'est pas satisfaite est retirée de l'expression du problème et génère 7 cas de figures différents. Les trois premiers correspondent au traitement de la sous-contrainte (1,1), les trois suivant à la sous-contrainte (1,2) et le dernier à la sous-contrainte (3,3).

1	$couleur(rouge)$	2	$couleur(vert)$	3	$couleur(bleu)$
4	$couleur(rouge)$, $\wedge couleur(vert)$	5	$couleur(rouge)$, $\wedge couleur(bleu)$	6	$couleur(vert)$, $\wedge couleur(bleu)$
7	$couleur(rouge) \wedge couleur(vert) \wedge couleur(bleu)$				

Le cas n° 3 mène au sous-problème suivant :

$$\begin{array}{ll} (c_1) & \neg couleur(X) \vee dominante(X) \\ (c_2) & \neg dominante(bleu) \\ (c_{3,3}) & couleur(bleu) \end{array}$$

qui est inconsistant. On a ainsi pu déterminer que le problème était inconsistant si l'objet était $bleu$.

Notons que cette *séparation de cas* augmente de manière prononcée la combinatoire du problème, d'autant plus que le nombre de contraintes non satisfaites est grand. On peut alors envisager d'utiliser des techniques heuristiques qui permettent de sélectionner les sous-ensembles de littéraux qui méneront à l'inconsistance. Il semblerait judicieux de *filtrer* les paquets de littéraux afin de déterminer ceux qui ont la plus forte probabilité de mener à une inconsistance et les tester en premier.

Pour l'exemple précédent, la résolution partielle du problème (pour les clauses (c_1) et (c_2)) aboutit à la génération des domaines suivants :

$$\mathcal{D} = \left\{ \begin{array}{ll} dom(\Psi_{couleur,\neg couleur}) & = \{\} \cup \{bleu\} \\ dom(\Psi_{dominante,\neg dominante}) & = \{\} \cup \{bleu\} \end{array} \right.$$

Un rapide examen de la clause (c_3) permet de déterminer que seul le littéral $couleur(bleu)$ produit une inconsistance.

6.7 Aspect Programmation en Logique

L'intérêt de la méthode de résolution des CSP^T_{ext} est qu'elle permet de rechercher à la fois une réfutation et un (ou des modèles) d'un problème. On peut donc envisager d'utiliser la méthode comme un système de Programmation en Logique. Cet aspect sera exposé de façon plus pragmatique dans le chapitre traitant des résultats car il s'agit d'une idée qui n'a germée que très récemment dans notre esprit. Notons, avant d'aller plus avant, qu'il ne s'agit pas de faire de la Programmation en Logique à la manière de Prolog.

Ainsi que nous l'avons mentionné dans l'introduction de cette thèse, l'une des raisons de l'efficacité de la Programmation en Logique provient de l'aspect procédural des clauses de Horn qui a pour effet de limiter l'espace de recherche.

En Démonstration Automatique et en particulier pour les problèmes exprimés sous forme normale conjonctive, il n'existe aucun moyen d'orienter la recherche. Dans le cadre de *Sacre*, on peut justement définir un mécanisme qui permette de focaliser la recherche dans une direction donnée ce qui conduit à réduire l'espace de recherche.

Nous avons vu que la transformation sous forme de CSP^T_{ext} consistait à tranformer une clause en toutes ses *contrapositives* :

$$p_1(t_1) \vee p_2(t_2) \vee \ldots \vee p_n(t_n) \equiv \left\{ \begin{array}{l} \neg p_1(t_1) \wedge \neg p_2(t_2) \wedge \ldots \wedge \neg p_{n-1}(t_{n-1}) \rightarrow p_n(t_n) \\ \vdots \\ \neg p_2(t_2) \wedge \neg p_3(t_3) \wedge \ldots \wedge \neg p_n(t_n) \rightarrow p_1(t_1) \end{array} \right.$$

ce qui a pour effet d'augmenter la combinatoire du problème et de détériorer les performances du chaînage avant.

Ainsi, pour la clause :

$$grandpere(X,Z) \vee \neg pere(X,Y) \vee \neg pere(Y,Z)$$

on obtient trois contrapositives :

$$\begin{cases} pere(X,Y) \wedge pere(Y,Z) & \rightarrow & grandpere(X,Z) \\ \neg grandpere(X,Z) \wedge pere(Y,Z) & \rightarrow & \neg pere(X,Y) \\ \neg grandpere(X,Z) \wedge pere(X,Y) & \rightarrow & \neg pere(Y,Z) \end{cases}$$

Il semble alors judicieux d'éliminer les clauses sources de redondance mais également de réduire l'ensemble précédent de contrapositives à un sous-ensemble minimum. Cela est effectivement possible pour des clauses du type $H \longrightarrow C$, où H est une conjonction de littéraux (appelés *hypothèses*) et C un seul littéral (la *conclusion*). Dans l'exemple précédent, il est évident qu'on désire uniquement déterminer les personnes qui sont grand-père d'une autre personne. La clause se résume en fait à :

$$pere(X,Y) \wedge pere(Y,Z) \longrightarrow grandpere(X,Z)$$

mais sa mise sous forme normale conjonctive fait disparaître la notion de déduction liée à cette clause.

Nous proposons alors dans le cadre de *Sacre* de fournir à l'utilisateur la possibilité de stipuler de telles clauses, dites clauses *orientées*, grâce à l'introduction du symbole `<-` (appelé symbole d'orientation) afin de préserver le caractère déductif de ce type de clauses. Ainsi, la clause précédente s'écrira :

```
grandpere(X,Z) <-  -pere(X,Y) |  -pere(Y,Z).
```

Remarquons que la notion de clause orientée est proche des notions relatives aux systèmes experts. Le symbole `<-` est assez proche de $:-$ de *Prolog*, à la différence qu'il n'y a aucune restriction quant au signe du littéral de tête. Elle autorise une résolution plus efficace des problèmes (cf. Résultats) en éliminant une partie de l'arbre de recherche par introduction d'un mécanisme de contrôle dans la spécification du problème :

> « *Exhaustively applying inference rules in an attempt to solve the problem is too naive an approach. The attempt to solve even the simplest of problems or questions or puzzles by exhaustive attack would produce vast amounts of information, much of it irrelevant. Considerable control of the application of reasoning rules is needed.* » ([132] p 5).

6.8 Classes de problèmes traités

La méthode *Sacre* est avant tout une méthode conçue pour réaliser de la preuve de théorème et de la déduction. Il s'agit, étant donné un problème, d'en démontrer la consistance. Si le problème est effectivement inconsistant, nous sommes alors en mesure d'en fournir la preuve. Par contre, si le problème est consistant, nous sommes capables d'exhiber des modèles du problème.

La méthode *Sacre* s'applique à l'ensemble des problèmes de la librairie TPTP[55] qui regroupe un ensemble de problèmes tests pour la Démonstration Automatique. Ces problèmes exprimés en logique du premier ordre sous forme normale conjonctive sont classés par catégories et concernent des domaines comme la logique (logique combinatoire COL, modèle de Henkin HEN), les mathématiques (théories des groupes GRP, des ensembles SET, la topologie TOP, la géométrie GEO) ou l'informatique (Conception CID et vérification CIV de circuits logiques). La version 1.2.1 de la librairie TPTP comporte plus de 2800 problèmes.

En outre, *Sacre* nous permet de résoudre certains problèmes relatifs à la Programmation en Logique, ou tout au moins d'attaquer ces problèmes selon une approche algorithmique, ce qui nous conduit à obtenir, au sortir de la résolution, le résultat d'un calcul et non plus seulement le caractère de consistance du problème considéré.

55. Thousands of Problems for Theorem Provers

6.9 Terminaison

En raison de la propriété de semi-décidabilité de la logique des prédicats du premier ordre, tout démonstrateur ne termine par forcément si la formule à démontrer n'est pas un théorème ou si l'ensemble de clauses à résoudre est consistant.

En ce qui concerne *Sacre*, nous sommes cependant en mesure de *programmer* la terminaison :

- soit en ayant recours à des procédés externes (voir implantation) :
  - fixer un temps limite de résolution, au dela duquel la recherche s'achève,
  - utiliser des heuristiques qui limitent la production de nouveaux faits (weighting),
- soit en ayant recours à un procédé interne qui consiste à fixer la taille limite des Ψ-domaines. Les domaines sont alors étendus jusqu'à ce que la taille qui leur a été fixée soit atteinte.

6.10 Conclusion

Nous avons montré qu'il était possible de représenter et de résoudre les problèmes de la logique des prédicats du premier ordre sous forme de Problèmes de Satisfaction de Contraintes appelés CSP^T_{ext}. La méthode qui découle de nos travaux modifie fortement le cadre CSP puisqu'elle se fonde sur une extension des domaines, alors que le cadre classique travaille sur des domaines de taille fixe. Nous avons également montré que l'équivalence CNF/CSP qui existait pour la représentation des problèmes, se retrouvait également au niveau de leur résolution.

Chapitre 7

Description du système *Sacre*

Nous décrivons dans ce chapitre les caractéristiques techniques du logiciel Sacre *et montrons sur quelques applications comment l'utiliser pour exprimer et résoudre des problèmes.*

7.1 Prototype

Un premier prototype de *Sacre* a été créé au commencement de cette thèse. Ce prototype était opérationnel fin Décembre 1996 et nous a permis de valider notre approche. Nous avons pu résoudre des problèmes élémentaires et expérimenter quelques idées de bases qui ont ensuite été reprises pour la conception du logiciel final.

Ce prototype a cependant nécessité une refonte car n'étant pas familier des techniques et structures de données employées afin de réaliser l'unification/désunification de deux termes, j'ai élaboré mon propre système de représentation et d'unification des termes logiques. Si celui-ci semblait attrayant par certains aspects, dont notamment un codage plus économique en mémoire pour le stockage des termes, cet avantage s'est révélé un défaut majeur lors de la réalisation de multiples unifications/désunifications de deux termes. Au cours de l'année 1997, nous avons eu connaissance de la mise à disposition d'un manuel expliquant les techniques de représentation des termes logiques ainsi que des procédures d'unification qui en découlent, utilisées dans la conception des démonstrateurs de théorèmes de la famille Otter (voir [17]). L'ensemble de ces structures et fonctions est écrit en langage C.

7.2 Implantation

Pour des raisons d'efficacité *Sacre* a été implanté en langage C et est compilable par le *gcc* sous Unix ou Linux. Les logiciels *lex* et *yacc* sont également requis. Le nombre de lignes de code est de plus de 15000 (commentaires inclus) et est très lisible.

Les structures utilisées pour la représentation des termes ainsi que les procédures d'unification sont tirées de [17]. Elles ont cependant été entièrement réécrites en raison de leur manque d'homogénéité. Elles constituent environ un quart du source.

7.3 Description générale du système

Le système *Sacre*[56] se compose :

- d'un démonstrateur automatique de théorèmes pour la logique des prédicats du premier ordre avec traitement de l'égalité implantant la méthode de résolution presentée dans cette thèse,
- d'une interface graphique chargée de simplifier l'utilisation de *Sacre* notamment dans la gestion des paramètres et heuristiques. Ce frontal est écrit en Java.

56. pour SAtisfaction de Contraintes et REsolution

7.4 Langage de modélisation

7.4.1 Conventions

Les conventions employées pour la représentation des symboles de constantes, de variables, de fonctions et de prédicats, sont les suivantes :

- les symboles de constantes, de prédicats et de fonctions doivent obligatoirement commencer par une lettre minuscules et sont suivis de caractères alphanumériques (lettres ou chiffres). Par exemple `p, q, f, sum, xA` sont des symboles valides,
- les noms de variables commencent par une lettre majuscule suivie par des caractères alphanumériques. Exemples : `X, X1, Xsmall, YA`.

7.4.2 Spécification en entrée

Sacre accepte en entrée une spécification qui peut être formulée en logique sous Forme Normale Conjonctive ou sous forme d'un Problème de Satisfaction de Contraintes (voir les figures 7.1 et 7.2). La grammaire de *Sacre* est fournie en annexe.

```
Problem Coloriage;

Clauses

  voisin( rouge, vert  ).
  voisin( rouge, bleu  ).
  voisin( vert,  rouge ).
  voisin( vert,  bleu  ).
  voisin( bleu,  rouge ).
  voisin( bleu,  vert  ).
  coloriage( X1, X2, X3, X4 ) | -voisin(X1,X2) | -voisin(X1,X4),
                              | -voisin(X2,X3) | -voisin(X2,X4) | -voisin(X3,X4).
Find coloriage(X1,X2,X3,X4).

List coloriage+.
```

FIG. 7.1 – *Problème de coloriage en Sacre : spécification logique*

7.4.3 Résultat en sortie

Le résultat de l'exécution du logiciel *Sacre* est influencée par deux directives utilisées dans la spécification du problème.

La première de ces directives est la commande `Find` suivie d'une liste de littéraux. Cette directive s'utilise afin d'afficher à l'écran les valeurs produites par la satisfaction des contraintes au cours de la résolution et qui répondent à certains critères. Par exemple pour le problème de coloriage précédent, l'instruction `Find coloriage(X1,X2,X3,X4).` signifie que l'ensemble des coloriages possibles seront affichés. Si l'on désire afficher uniquement les coloriages pour lesquels $X1$ est rouge ou vert, on écrira :

```
Find coloriage(rouge,X2,X3,X4), coloriage(vert,X2,X3,X4).
```

La deuxième directive est `List`. Elle permet, une fois la résolution effectuée, d'afficher les domaines des variables. Pour le problème de coloriage, l'instruction `List coloriage+.` affichera l'ensemble des solutions trouvées pendant la résolution.

```
Problem Coloriage;

Domains
  voisin+ = { ( rouge, vert ),  ( rouge, bleu ),
              ( vert,  rouge ), ( vert,  bleu ),
              ( bleu,  rouge ), ( bleu,  vert ) }

Constraints
  #(1,6, coloriage( X1, X2, X3, X4 ), -voisin(X1,X2), -voisin(X1,X4),
                                      -voisin(X2,X3), -voisin(X2,X4),
                                      -voisin(X3,X4) ).

Find coloriage(X1,X2,X3,X4).

List coloriage+.
```

FIG. 7.2 – *Problème de Coloriage en Sacre : spécification CSP*

7.5 Spécificités du logiciel

Le logiciel *Sacre* implante un certain nombre de techniques issues de la Démonstration Automatique, mais également de la Programmation par Contraintes, parmi lesquelles (voir également C) :

– En Démonstration Automatique :

- des techniques d'élimination des termes redondants comme la subsumption et la backsubsumption (`-back-subsumption`),
- des techniques de limitations des termes produits, comme le *weighting* (`-weight`) ou l'élimination des termes dont l'arité est supérieure à une valeur limite (`-arity`),
- des stratégies de parcours en largeur d'abord, en profondeur d'abord ou par approfondissements successifs (`-cover`),
- la démodulation (`-demodulation`) avec ou sans LRPO [34] pour le traitement de l'égalité (`-lrpo`),

– En Programmation Par Contraintes :

- des stratégies de parcours basées sur des heuristiques de choix de variables et de contraintes (`-heuristic`),
- des techniques de limitations des termes produits, qui se basent sur la limitation du nombre de valeurs produites lors de la satisfaction d'une contrainte (`-deductions`) ou la définition d'une taille fixe pour certains domaines (instruction `Of Size`),
- des techniques chargées d'augmenter l'efficacité des traitements et qui ont pour objet de limiter le nombre de contraintes produites lors de la traduction du problème logique en CSP (`-freeze` et `-horn`),
- une technique de classement des valeurs dans les domaines pour une recherche d'unificateurs accélérée.

7.5.1 Algorithmes

L'algorithme combinant recherche de consistance et calcul de modèles est donné figure 7.3 :

il représente la traduction des règles de propagation et d'inconsistance décrites formellement dans le chapitre précédent.

fonction $Consistance(\mathcal{P} : CSP^T_{ext})$: *booléen*
entree : un CSP^T_{ext}
sortie : CSP^T_{ext} dont les domaines ont été étendus
retourne : *vrai* si le problème est consistant,
faux sinon
debut
$V = \{\Psi_p \in \mathcal{V}, dom^N(\Psi_p) \neq \emptyset\}$
$consistance = vrai$
tantque $V \neq \emptyset$ et $consistance$ faire
choisir $\Psi_v \in V$
$V = V/\{\Psi_v\}$
$C(\Psi_v) = \{\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_{n-1}}^{[t_{n-1}]}, \Psi_v^{[t_v]} >\}$
tantque $C(\Psi_v) \neq \emptyset$ et $consistance$ faire
choisir $c \in C(\Psi_v)$
$C(\Psi_v) = C(\Psi_v)/\{c\}$
$consistance = propagation(V,c,\Psi_v)$
ftq
ftq
retourne $consistance$
fin

FIG. 7.3 – *Algorithme de recherche de consistance en Sacre.*

7.5.2 Architecture du système

L'architecture du système est donnée figure 7.5 et s'organise autour de deux modules principaux :

- le *module de traduction* a pour but de traduire le problème initial exprimé sous forme CNF en un CSP^T_{ext} équivalent. Il effectue également une vérification syntaxique et sémantique de l'expression du problème,
- le *module de satisfaction*, qui constitue le coeur du système de résolution, consiste à organiser la satisfaction des contraintes en fonction de l'évolution des domaines et des paramètres définis pour résoudre le probème.

7.6 Couverture de l'espace de recherche

Un espace de recherche du type arbre peut être couvert suivant au moins trois stratégies différentes (voir figure 7.6) :

- la recherche en **profondeur d'abord** consiste à examiner les fils du noeud le plus récemment développé appelé le noeud père. Lorsqu'un noeud ne peut pas être développé plus avant, on effectue un *retour arrière*, c'est-à-dire, que l'on remonte d'un niveau dans l'arbre, et on examine les autres fils issus du noeud père. L'inconvénient de la recherche en profondeur d'abord est sa complexité en temps, car toutes les branches d'un fils d'un noeud père doivent être explorées avant de passer à l'exploration des autres fils du noeud père,
- la recherche en **largeur d'abord** consiste à développer tous les noeuds situés à la même profondeur avant de développer les noeuds à une profondeur supérieure. L'inconvénient de la recherche en largeur d'abord est sa complexité en espace, celle-ci est exponentielle car il est nécessaire de stocker les noeuds jusqu'à la profondeur d avant d'examiner les noeuds à la profondeur $d+1$. La complexité d'un parcours en largeur d'abord jusqu'à la profondeur d, pour un facteur de branchement b, est donné par la formule :

fonction *propagation*(V : un ensemble de Ψ-variables
c : une contrainte $\# < 1, n, \Psi_{p_1}^{[t_1]}, \ldots, \Psi_{p_{n-1}}^{[t_{n-1}]}, \Psi_v^{[t_v]} >$
Ψ_v : une variable) : *booléen*
sortie : $vrai$ si le problème est consistant, $faux$ sinon
debut
$U_f(\Psi_v) = \{(\omega,\mu) \in U_f(\Psi_{\neg v}^{[t_v]}) \quad \text{tels que} \quad |\Psi_{\neg v}^{[\omega(t_v)]}| = 0\}$
pour chaque $(\omega,\mu) \in U_f(\Psi_v)$ faire
$\Sigma(\Psi_v) = \Sigma(\# < 1, 1, \Psi_{p_1}^{[\sigma(t_1)]}, \ldots, \Psi_{p_n}^{[\sigma(t_n)]}, \Psi_{\neg v}^{[\omega(t_v)]} >$
pour chaque $\sigma \in \Sigma(\Psi_v)$ tel que $|\Psi_{p_i}^{[\sigma(t_i)]}| = 1$ faire
$\nu = \sigma(t_i)$
if $\nu \in dom(\Psi_{\neg p_i})$ alors retourne faux
$dom^N(\Psi_{p_i}) = dom^N(\Psi_{p_i}) \cup \{\sigma(t_i)\}$
if $\Psi_{p_i} \notin V$ alors $V = V \cup \{\Psi_{p_i}\}$
fpour
fpour
retourne $vrai$
fin

FIG. 7.4 – *Algorithme de propagation d'une contrainte.*

$$\begin{aligned} compl_{larg}(d,b) &= \textstyle\sum_{i=1}^{d} b^i \\ &= \tfrac{1}{b-1} \textstyle\sum_{i=1}^{d} (b^{i+1} - b^i) \\ &= \tfrac{b}{b-1}(b^d - 1) \end{aligned}$$

- le recherche **par approfondissements successifs** [61] combine recherche en profondeur d'abord et recherche en largeur d'abord. Elle consiste à effectuer des recherches en profondeur d'abord jusqu'à une profondeur limite fixée. Si aucune solution n'est trouvée, on réitère la recherche en augmentant la valeur de la profondeur limite. La recherche par approfondissements successifs peut se révéler plus efficace que les deux premières couvertures. D'une part, s'il existe une solution à une profondeur donnée, on n'examinera pas les noeuds de profondeur supérieure, ce qui évite l'enlisement dans le parcours de branches infinies. D'autre part, le surcoût en temps est faible, la complexité pour ce mode de couverture étant égale à :

$$\begin{aligned} compl_{approf}(d,b) &= \textstyle\sum_{i=1}^{d} compl_{larg}(d,b) \\ &= \textstyle\sum_{i=1}^{d} \tfrac{b}{b-1}(b^i - 1) \\ &= \tfrac{b}{b-1}\left(\textstyle\sum_{i=1}^{d}(b^i) - d\right) \\ &= \tfrac{b}{b-1}\left(compl_{larg}(d,b) - d\right) \end{aligned}$$

Dans le cadre de *Sacre*, un noeud représente un ensemble de contraintes en relation avec une variable et une branche est un choix de variable. Le choix de la couverture est réalisé grâce à l'option en ligne de commande `-cover`. Celle-ci requiert un argument choisi parmi la liste suivante :

`depth`	pour une recherche en profondeur d'abord,
`breadth`	pour une recherche en largeur d'abord,
`iterative`	pour une recherche par approfondissements successifs.

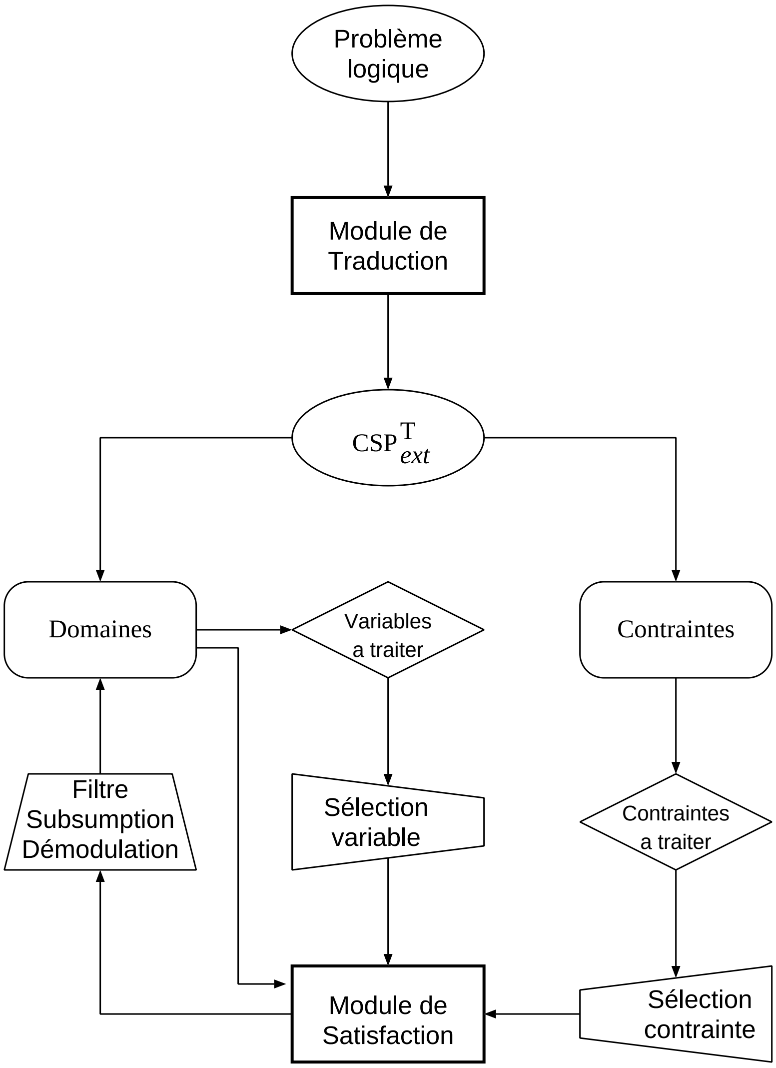

FIG. 7.5 – *Architecture de Sacre.*

7.7 Applications

Dans cette section nous montrons comment spécifier et résoudre des problèmes avec *Sacre*.

7.7.1 Aspect Démonstration Automatique

Nous commençons par nous intéresser à l'aspect Démonstration Automatique, pour lequel on cherche à prouver la consistance ou l'inconsistance d'un ensemble d'axiomes.

7.7.1.1 Théorie des groupes

Considérons le problème de la figure 7.7. Ce problème est tiré de [23] et est généralement référencé sous l'appellation Chang et Lee $n°$ 2.

On veut démontrer que dans un groupe $(E,\circ)$, si chaque élément est son propre inverse alors le groupe est commutatif. Pour exprimer ce problème sous forme CNF, on introduit le prédicat $p(X,Y,Z)$ qui s'interprète comme $X \circ Y = Z$. On traduit ensuite les différents axiomes du problème numérotés sur la figure 7.7 : élément neutre à gauche (a_1) et à droite (a_2), produit d'un élément par lui même égal

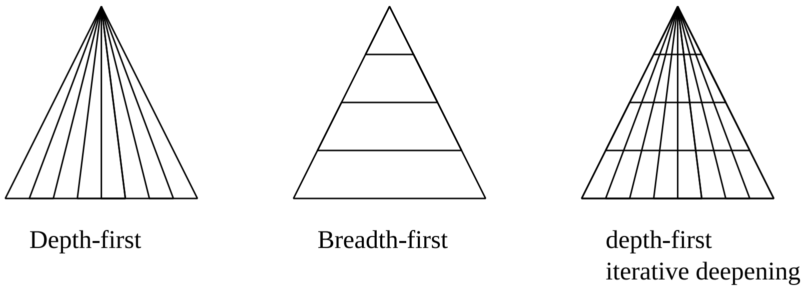

FIG. 7.6 – *Représentation schématique des stratégies de parcours d'un arbre de recherche*

```
     Problem ChangLee2;

     Clauses
(a1)   p(e,X,X).
(a2)   p(X,e,X).
(a3)   p(X,X,e).
(a4)   p(U,Z,W) | -p(X,Y,U) | -p(Y,Z,V) | -p(X,V,W).
(a5)   p(X,V,W) | -p(X,Y,U) | -p(Y,Z,V) | -p(U,Z,W).
(a6)   p(a,b,c).
(a7)   -p(b,a,c).
```

FIG. 7.7 – *Problème* changlee2.pb.

à l'élément neutre (a_3), associativité (a_4,a_5). Le but de ce problème est de vérifier la consistance du système par réfutation. La démarche employée est classique en démonstration automatique de théorèmes et consiste à démontrer l'inconsistance de l'ensemble d'axiomes (1-5) augmenté de la négation de la conclusion (a_6,a_7) : $\exists a,b,c \in E$ tels que $(a \circ b = c) \wedge (a \circ b \neq c)$.

La résolution de ce problème est conduite par la commande suivante :

```
sacre changlee2.pb
```

Le résultat de l'exécution de *Sacre* sur un problème de ce type est une réponse booléenne. Le problème étant insatisfiable on obtient la réponse : UNSATISFIABLE.

7.7.1.2 Théories égalitaires

Un certain nombre de problèmes en Démonstration Automatique font partie du domaine des mathématiques et utilisent la relation d'égalité. *Sacre* offre la possibilité d'exprimer et de traiter par la *démodulation* avec ou sans LRPO les problèmes égalitaires. Pour exprimer l'égalité entre deux termes, soit $t_1 = t_2$, on utilise le prédicats à deux arguments $equal(t_1,t_2)$.

Un exemple de problème faisant appel au concept d'égalité est le problème *BOO016-1.pb* de la compétition CADE-13. Il consiste à démontrer que dans une algèbre de Boole, le produit et la somme sont des opérations liées, en d'autres termes, on veut montrer que si $\exists,a,b,c$ tels que $a \times b = c$ alors $a + c = a$. Pour résoudre ce type de problème on utilise la commande suivante :

```
sacre -auto BOO016-1
```

Le problème *BOO016-1.pb* est un exemple de problème que *Sacre* ne permet pas de résoudre sans avoir recours à des heuristiques qui limitent l'espace de recherche en réduisant le nombre de clauses générées lors de la transformation en CSP. Le mode `-auto` (pour automatique) est une combinaison d'heuristiques qui donnent en général de très bons résultats sur la plupart des problèmes de la compétition CADE-13. Il est équivalent à la ligne d'options suivante :

```
-demodulation -lrpo left -weight 5 -arity 4 -deductions 1000 -freeze
```

Sacre démontre l'inconsistance de ce problème et prouve ainsi la conjecture énoncée.

7.7.1.3 La factorielle

Le problème de la factorielle est spécifié en logique des prédicats sous forme symbolique. Aucune interprétation des termes n'est effectuée car *Sacre* n'est pour l'instant pas en mesure d'interpréter les termes.

```
       Problem Factorielle;

       Clauses

(a1)     sum(X, zero, X).
(a2)     sum(X, s(Y), s(Z)) | -sum(X, Y, Z).
(a3)     product(s(zero), X, X).
(a4)     product(s(X), Y, Z) | -sum(R, Y, Z) | -product(X, Y, R).
(a5)     factorial(zero, s(zero)).
(a6)     factorial(s(X), Y) | -factorial(X, Z) | -product(s(X), Z, Y).
(a7)    -factorial( s(s(s(zero))), Results).
```

FIG. 7.8 – *Problème* NUM283-1.pb.

Le nombre 0 est représenté par le symbole de constante `zero`. Le symbole de fonction `s` représente quant à lui l'opération successeur. Ainsi, le nombre 1 a pour représentant `s(zero)`. L'inconsistance du problème précédent est obtenue par l'exécution de la commande suivante :

```
sacre NUM283-1.pb
```

7.7.2 Aspect Programmation en Logique

Nous passons à présent à l'examen de l'aspect Programmation en Logique du système de résolution.

7.7.2.1 Coloriage de carte

Le problème de la figure 7.9 consiste à colorier une carte avec trois couleurs rouge, vert, bleu, de manière à ce que deux rectangles adjacents ne soient pas de la même couleur. Résoudre ce problème consiste à exhiber les coloriages qui répondent à la spécification donnée.

L'instruction `Find` permet de spécifier que l'on cherche à déterminer les interprétations de la forme prédicative $coloriage(x_1,x_2,x_3,x_4)$, ce qui en prolog correspond à l'interrogation

$$?coloriage(x_1,x_2,x_3,x_4).$$

Dès qu'un modèle est trouvé, il est affiché, puis le système demande à l'utilisateur s'il doit continuer (`;`) ou s'arrêter (`q`). Si l'utilisateur décide de continuer la recherche, le système tentera de trouver de nouvelles interprétations. Pour résoudre le problème de coloriage de carte, on utilise la ligne de commande suivante :

```
sacre -v 1 colorie1
```

```
Problem Coloriage;

Clauses

  voisin( rouge, vert  ).
  voisin( rouge, bleu  ).
  voisin( vert,  rouge ).
  voisin( vert,  bleu  ).
  voisin( bleu,  rouge ).
  voisin( bleu,  vert  ).
  coloriage( X1, X2, X3, X4 ) | -voisin(X1,X2) | -voisin(X1,X4),
             | -voisin(X2,X3) | -voisin(X2,X4) | -voisin(X3,X4).

Find coloriage(X1,X2,X3,X4).
```

FIG. 7.9 – *Problème* colorie1.pb

où v correspond à l'option *verbose*. Les interprétations exhibées sont alors les suivantes :

```
=> (8) colorie(bleu,vert,bleu,rouge)  (; or q) ? ;
=> (9) colorie(bleu,rouge,bleu,vert)  (; or q) ? ;
=> (10) colorie(vert,rouge,vert,bleu)  (; or q) ? ;
=> (11) colorie(vert,bleu,vert,rouge)  (; or q) ? ;
=> (12) colorie(rouge,vert,rouge,bleu)  (; or q) ? ;
=> (13) colorie(rouge,bleu,rouge,vert)  (; or q) ? ;
```

7.7.2.2 Schubert's steamroller

Le "*rouleau compresseur*" de Schubert est un problème très simple puisqu'il ne comporte aucun symbole de fonction, il est donc à domaines finis.

Ce problème ne pose aucune difficulté à la grande majorité des démonstrateurs de théorèmes. Cependant, comme l'a montré **Ramsay** [99], une légère modification de ce problème qui correspond à l'ajout, aux axiomes initiaux du problème, de la clause :

```
-animal(X) | -animal(Y) | quicker(X,Y) | smaller(Y,X)
```

peut entraîner une dérive du démonstrateur de théorèmes *Satchmo* [76] causée par une explosion combinatoire. Cette dérive tient à la méthode de résolution employée par *Satchmo* (voir [70] pour de plus amples explications). *Sacre* ne rencontre aucune difficulté à résoudre ce problème :

```
sacre -v 2  steam
```

7.7.2.3 La factorielle revisitée

Reprenons le problème de la factorielle auquel on adjoint l'instruction `List factorial+` (voir figure 7.11). Si l'on cherche à résoudre ce problème suivant une démarche Démonstration Automatique, seule nous intéresse la consistance des axiomes. Par contre, dans une optique de Programmation en Logique, nous cherchons à obtenir un modèle ou tout au moins un résultat à la question (a_7).

Pour résoudre ce type de problème, on utilisera l'option `-horng` qui transforme les axiomes (a_2), (a_4) et (a_6) en clauses de Horn et permet de focaliser la recherche de manière à fournir un résultat au calcul requis par l'axiome (a_7). On résout alors le problème par :

```
sacre -horn factorial
```

```
Clauses
 -wolf(X) | -fox(Y)   | -likes(X,Y).
 -wolf(X) | -grain(Y) | -likes(X,Y).
 -bird(X) | -snail(Y) | -likes(X,Y).
 -animal(X) | -animal(Y) | -likes(X,Y) | -grain(Z) | -likes(Y,Z).

  wolf(w).
  fox(f).
  bird(b).
  snail(s).
  caterpillar(c).
  grain(g).

  animal(X) | -wolf(X).
  animal(X) | -fox(X).
  animal(X) | -bird(X).
  animal(X) | -snail(X).
  animal(X) | -caterpillar(X).
  plant(X)  | -grain(X).
  plant(i(X)) | -snail(X).
  plant(h(X)) | -caterpillar(X).

  smaller(X,Y) | -caterpillar(X) | -bird(Y).
  smaller(X,Y) | -snail(X) | -bird(Y).
  smaller(X,Y) | -bird(X) | -fox(Y).
  smaller(X,Y) | -fox(X) | -wolf(Y).

  likes(X,Y) | -bird(X) | -caterpillar(Y).
  likes(X,i(X)) | -snail(X).
  likes(X,h(X)) | -caterpillar(X).

  likes(X,Y) | likes(X,Z) | -animal(X) | -animal(Y) | -smaller(Y,X) |
        -plant(W) | -likes(Y,W) | -plant(Z).
```

FIG. 7.10 – *Problème* steam.pb.

Le résultat en sortie de ce problème est, d'une part, l'inconsistance du problème, et d'autre part, l'ensemble de littéraux produits durant la résolution et obtenus grâce à la directive `List factorial+` :

```
[+factorial] = {
       factorial(zero,s(zero)),
       factorial(s(s(zero)),s(s(zero))),
       factorial(s(zero),s(zero)),
       factorial(s(s(s(zero))),s(s(s(s(s(s(zero))))))
               }
```

En particulier, le dernier littéral correspond au résultat du calcul demandé, soit $3! = 6$.

On peut également désirer déterminer si le nombre 6 correspond à l'évaluation de la factorielle d'un nombre N. Dans ce cas, il faut remplacer l'axiome (a_7) par l'axiome suivant :

```
-factorial( N, s(s(s(s(s(s(zero)))))) ).
```

```
        Problem Factorielle;

        Clauses

(a1)      sum(X, zero, X).
(a2)      sum(X, s(Y), s(Z)) | -sum(X, Y, Z).
(a3)      product(s(zero), X, X).
(a4)      product(s(X), Y, Z) | -sum(R, Y, Z) | -product(X, Y, R).
(a5)      factorial(zero, s(zero)).
(a6)      factorial(s(X), Y) | -factorial(X, Z) | -product(s(X), Z, Y).
(a7)     -factorial( s(s(s(zero))), Results).

          List factorial+.
```

FIG. 7.11 – *Problème* factorial.pb.

```
Problem Blocs

Clauses

#(1,1, couleur(a,rouge), couleur(a, vert)).
couleur(b,rouge).
#(1,1,couleur(c, vert), couleur(c, bleu)).
couleur(d, bleu).

pile(X,CX,Y,CY,Z,CZ) | -couleur(X,CX) | -couleur(Y,CY) | -couleur(Z,CZ).

Find pile(X,rouge,Y,vert,Z,bleu).
```

FIG. 7.12 – *Problème* blocs.pb.

7.7.2.4 Traitement des informations incomplètes

Le problème de la figure 7.12 est un exemple de problème que *Sacre* est pour l'instant dans l'incapacité de traiter car il exprime des informations incomplètes que *Sacre* ne peut traiter que par l'intermédiaire d'une procédure de *séparation par cas*.

On dispose de quatre blocs a,b,c,d. On sait que b est rouge, d est bleu, a est rouge ou vert, c est vert ou bleu. Ces blocs peuvent être empilés et on cherche à savoir combien on peut former de piles de trois blocs telles que le premier bloc soit rouge, le deuxième soit vert et le troisième soit bleu (instruction `Find`).

Lorsque la *séparation de cas* sera implantée dans *Sacre*, nous serons à même de résoudre ce problème et d'obtenir les réponses suivantes :

```
pile(b,rouge,c,vert,d,bleu)
pile(a,rouge,c,vert,d,bleu)
pile(b,rouge,a,vert,d,bleu)
pile(b,rouge,c,vert,d,bleu)
pile(b,rouge,a,vert,c,bleu)
pile(b,rouge,a,vert,d,bleu)
```

7.8 Champ d'application

Le logiciel *Sacre* est actuellement capable de résoudre les problèmes suivants :

- les problèmes définis, c'est-à-dire composés d'un ensemble de clauses de Horn et ne spécifiant pas d'informations incomplètes. Cette catégorie englobe une partie des problèmes de la Programmation en Logique.

- les problèmes avec égalité susceptibles d'être résolus uniquement en utilisant la démodulation,

Concernant les problèmes de la Programmation en Logique, *Sacre* n'est pas toujours capable d'évaluer les *réponses correctes* comme le ferait *Prolog*. L'exemple de la *factorielle* exposé précédemment est un exemple de problème pour lequel *Sacre* parvient à exhiber une réponse correcte. Par contre, pour la concaténation de listes (cf. fig. 7.13), on ne parvient pas à obtenir la réponse souhaitée :

```
concat([a.b.nil],[c.nil],[a.b.c.nil])
```

```
Problem Concatenation;

Clauses

  concat([],L,L).
  concat(L,[],L).
  concat([X.L],M,[X.N]) | -concat(L,M,N)).
 -concat([a.b.nil],[c.nil],Z).
```

FIG. 7.13 – *Problème* liste1.pb

7.9 Conclusion

Nous avons montré que *Sacre* nous permettait d'exprimer dans un langage déclaratif et statique des problèmes relatifs à la Démonstration Automatique et à la Programmation en Logique. Le problème de la factorielle est sans doute l'exemple le plus remarquable illustrant pleinement la différence qui existe entre la Démonstration Automatique et la Programmation en Logique.

L'expression de ce problème est en effet la même en Démonstration Automatique ou en Programmation en Logique, hormis l'instruction `Find` utilisée dans l'expression du problème *factorial.pb*, seule change l'approche de résolution. Si l'on cherche à démontrer la consistance d'un ensemble d'axiomes, on aura tendance à explorer une grande partie de l'espace de recherche en ayant cependant le souci de limiter l'explosion combinatoire. En revanche, si l'on cherche à *calculer* un résultat, il importe d'explorer l'espace de recherche en se focalisant dans la direction menant au calcul.

Chapitre 8

Expérimentations et Résultats

Ce chapitre a pour but d'évaluer les performances de Sacre *et de situer celui-ci par rapport aux systèmes existants. L'ensemble des résultats présentés dans ce chapitre ont été obtenus sur un Pentium^{TM} 200 MMX sous Linux.*

8.1 Résultats en Démonstration Automatique

Nous présentons dans cette section les résultats obtenus par *Otter*, *Setheo* et *Sacre* pour la résolution des problèmes de la compétition CADE-13 [119].

8.1.1 Otter et Setheo

La question que l'on peut être amené à se poser d'emblée est la suivante : pourquoi comparer *Sacre* à *Otter* et *Setheo* ?

Otter et *Setheo* sont deux démonstrateurs de théorèmes éprouvés. Ils se sont montrés les plus performants et ont obtenu les meilleurs résultats lors de la compétition CADE-13 (voir section 3.7).

Les résultats que nous présentons table 8.2 ne sont pas l'exact reflet des résultats de la compétition CADE-13 proposés dans [119] et pour laquelle les démonstrateurs avaient été adaptés en fonction des problèmes à résoudre. *Otter* est donc ici utilisé dans sa version 3.0.2 de Juin 1994 et *Setheo* dans sa version 3.3.

Pour *Otter*, nous avons fait figurer les résultats obtenus avec le mode autonome (colonne HR) qui utilise l'hyperrésolution, ainsi que les résultats obtenus avec la résolution unitaire (colonne UR). En ce qui concerne *Setheo*, nous présentons les résultats qui correspondent aux stratégies ***tableau depth*** (colonne DR) et ***weighted depth*** (colonne WDR). Les résultats ont été arrondis à l'ordre supérieur, cependant une valeur de 0 signifie que la résolution s'est effectuée en moins de 0.5 secondes.

		Pb Résolus	Temps	Efficacité
Otter	HR	31	607	1.58
Otter	UR	29	516	1.63
Setheo	DR	26	279	2.42
Setheo	WDR	29	817	1.03

TAB. 8.1 – *Efficacité de Otter et Setheo*

L'efficacité des démonstrateurs *Otter* et *Setheo*, comme définie section 3.7 est présentée table 8.1. *Setheo* est le plus efficace lorsqu'il utilise la stratégie (DR).

Problème	Otter		Setheo		Problème	Otter		Setheo	
	HR	UR	DR	WDR		HR	UR	DR	WDR
BOO006-1	3	1	5	0	PLA014-1	-	-	0	0
BOO012-1	8	1	0	0	PLA014-2	-	-	0	-
BOO016-1	2	0	-	11	PLA018-1	-	-	1	-
CAT003-4	161	0	0	0	RNG005-1	1	1	-	199
CAT012-3	42	40	0	2	RNG038-2	0	0	-	94
COL002-2	3	3	0	0	RNG040-1	0	0	0	0
COL002-3	-	-	0	0	SET008-1	0	0	0	0
COL003-4	-	-	13	101	SET015-2	-	-	-	-
GEO002-1	-	-	-	-	SET024-6	0	0	0	0
GEO006-1	-	-	-	-	SET061-6	16	3	-	-
GEO027-2	-	-	-	-	SET063-6	6	82	-	-
GEO059-2	-	-	-	152	SET075-6	42	54	-	
GRP048-2	0	0	-	39	SET080-6	0	0	-	3
GRP099-1	83	82	-	-	SET082-6	75	76	-	-
HEN003-1	0	-	-	-	SET083-6	76	98	237	-
LCL196-1	7	21	6	6	SET101-6	0	0	0	0
LCL210-1	4	43	6	7	SET196-6	69	-	1	0
NUM003-1	0	0	1	0	SET232-6	-	-	-	21
NUM009-1	8	9	3	152	SET240-6	-	-	-	30
NUM046-1	-	-	-	-	SET243-6	-	-	-	-
NUM159-1	-	-	-	-	SET261-6	-	-	-	-
NUM232-1	-	-	-	-	SET558-6	0	0	-	-
NUM284-1.010	0	1	-	-	SYN200-1	0	0	0	0
PLA004-1	-	-	0	-	SYN202-1	1	0	0	0
PLA011-2	-	-	0	0	SYN271-1	0	1	0	0

TAB. 8.2 – *Résultats d'Otter et Setheo pour les problèmes de la compétition CADE-13*

8.1.2 *Sacre*

L'ensemble des résultats obtenus par *Sacre* pour les problèmes de la compétition CADE-13 est présenté table 8.4. Nous avons voulu montrer l'influence sur le temps de résolution des problèmes des différentes techniques et heuristiques disponibles en *Sacre*. La colonne *Sans* correspond à une résolution des problèmes sans aucune heuristique, nous permettant ainsi d'identifier les problèmes les plus simples. Les colonnes listées ci-dessous correspondent à l'utilisation des heuristiques suivantes :

- *demod* : démodulation avec LRPO (lex recursive path ordering) [34],
- *freeze* : limitation du nombre de clauses produites grâce à l'option `-freeze`,
- *a4/w5* : élimination des termes de poids supérieur à 5 et d'arité supérieure à 4,
- *d1000* : limitation du nombre de valeurs produites par chaque contrainte aux 1000 premières valeurs,
- *auto* : combinaison des 4 colonnes précédentes.

La dernière colonne correspond à des cas de résolution particuliers pour certains problèmes :

† `-arity 6` uniquement
‡ `-arity 5` uniquement
§ `-horn` uniquement

Si l'on considère les problèmes résolus suivant l'heuristique `-auto`, l'efficacité de *Sacre* est de 22.04, ce qui représente une efficacité très nettement supérieure à celle d'*Otter* ou *Sehteo*.

On peut d'ailleurs noter (voir table 8.3) qu'*Otter*, *Setheo* et *Sacre* résolvent l'ensemble des problèmes présentés dans [23]

Problème	Otter	Setheo	Sacre
changlee1.pb	0.02	0.01	0.01
changlee2.pb	0.01	0.07	0.03
changlee3.pb	0.01	0.03	0.01
changlee4.pb	0.01	0.01	0.01
changlee5.pb	0.01	0.03	0.01
changlee6.pb	0.01	0.02	0.03
changlee7.pb	0.01	0.02	0.01
changlee8.pb	0.01	0.02	0.01
changlee9.pb	0.01	0.03	0.03

TAB. 8.3 – *Résultats comparatifs entre Otter, Setheo et* Sacre *pour les propblèmes Chang et Lee.*

Problème	Sans	demod	freeze	a4/w5	d1000	auto	Spécial
BOO006-1	0.05	0.65	0.38	0.65	2.14	0.13	
BOO012-1	107.60	58.26	118.94	0.86	2.96	0.40	
BOO016-1						0.56	
CAT003-4							
CAT012-3							
COL002-2							
COL002-3							
COL003-4							
GEO002-1							
GEO006-1							
GEO027-2							
GEO059-2							
GRP048-2						3.00	
GRP099-1							
HEN003-1							
LCL196-1							23.27†
LCL210-1							2.89‡
NUM003-1		6.84	6.83	0.04		0.02	
NUM009-1		0.27	0.26	0.15		0.13	
NUM046-1						2.30	
NUM159-1							
NUM232-1						0.28	
NUM284-1.010	2.39						0.04§
PLA004-1							
PLA011-2							
PLA014-1							
PLA014-2							
PLA018-1							
RNG005-1				39.20	1.37	0.47	
RNG038-2					35.60	8.11	
RNG040-1		0.01	0.01	0.02	0.01	0.01	
SET008-1	0.01	0.01		0.01	0.01	0.01	
SET015-2							
SET024-6	0.04	0.06	0.05	0.06	0.08	0.06	
SET061-6						1.13	
SET063-6				13.06		0.55	
SET075-6						1.96	
SET080-6	1.03		4.98	0.23		0.14	
SET082-6						1.51	
SET083-6				1.95		0.79	
SET101-6						5.73	
SET196-6	0.05		0.03	0.05	0.07	0.06	
SET232-6				4.41		0.64	
SET240-6	0.05		0.09	0.06	0.12	0.04	
SET243-6						5.71	
SET261-6				1.21		0.60	
SET558-6							
SYN200-1	0.09		0.11		0.11	0.09	
SYN202-1	0.36		0.46		0.42	0.44	
SYN271-1	0.67		0.89		0.69	0.70	

TAB. 8.4 – *Résultats de* Sacre *pour les problèmes de la compétition CADE-13*

Problème	Otter	Setheo	*Sacre*
BOO006-1	3	0	0
BOO012-1	8	0	0
BOO016-1	2	11	1
CAT003-4	161	0	
CAT012-3	42	2	
COL002-2	3	0	
COL002-3		0	
COL003-4		101	
GEO002-1			
GEO006-1			
GEO027-2			
GEO059-2		152	3
GRP048-2	0	39	
GRP099-1	83		
HEN003-1	0		
LCL196-1	7	6	
LCL210-1	4	7	
NUM003-1	0	0	0
NUM009-1	8	152	0
NUM046-1			2
NUM159-1			
NUM232-1			0
NUM284-1.010	0		
PLA004-1			
PLA011-2		0	
PLA014-1		0	
PLA014-2			
PLA018-1			
RNG005-1	1	199	0
RNG038-2	0	94	8
RNG040-1	0	0	0
SET008-1	0	0	0
SET015-2			
SET024-6	0	0	0
SET061-6	16		1
SET063-6	6		1
SET075-6	42		2
SET080-6	0	3	0
SET082-6	75		2
SET083-6	76		1
SET101-6	0	0	6
SET196-6	69	0	0
SET232-6		21	1
SET240-6		30	0
SET243-6			6
SET261-6			1
SET558-6	0		
SYN200-1	0	0	0
SYN202-1	1	0	0
SYN271-1	0	0	1

TAB. 8.5 – *Résultats comparatifs Otter, Setheo, Sacre*

8.1.3 Comparaison avec d'autres systèmes

Nous aurions aimé comparer *Sacre* à d'autres systèmes, notamment *Satchmo* et *CLIN-S*. *Satchmo* est un démonstrateur relativement proche de *Sacre*, tout au moins dans l'esprit. *CLIN-S* implante l'*hyperlinking* qui consiste à déterminer un sous-ensemble de clauses susceptibles de prouver l'inconsistance. Cependant, ces deux systèmes sont implantés en *Prolog* et il nous paraît difficile de conduire une comparaison avec notre système implanté en langage C. D'autre part, *Satchmo* et *CLIN-S* ont obtenu des résultats assez médiocres sur les problèmes de la compétition CADE-13. Leurs efficacités respectives sont de 0.68 et 0.07.

8.1.4 Analyse

En ce qui concerne les problèmes de la compétition CADE-13, *Sacre* se montre assez proche d'*Otter* car il résoud les classes de problèmes résolues par ce dernier excepté pour les problèmes de la famille *CAT* (Mathématiques) et le problème *COL002-2* (Logique) pour lesquels *Otter* utilise la *paramodulation* qui n'est pas implantée dans *Sacre*. Cette similitude par rapport aux classes de problèmes résolues tient probablement au fait que *Sacre* implante des mécanismes et techniques issus des démonstrateurs de la famille *Otter* (cf. [17]).

Sacre et *Otter* sont incapables de résoudre les problèmes de la catégorie *PLA* (Planification), alors que *Setheo* le peut. Il serait intéressant de vérifier si ces problèmes sont susceptibles d'être résolus suivant une approche Programmation en Logique en orientant certaines clauses. La difficulté liée à une telle démarche réside dans le choix des clauses qui seront orientées et dans le choix du littéral sur lequel portera l'orientation.

	Règle d'inférence	Traitement de l'égalité	Complétude
Otter	hyperrésolution	Paramodulation	complet
Setheo	model elimination	Paramodulation	complet
Sacre	résolution unitaire	Demodulation	incomplet

TAB. 8.6 – *Caractéristiques d'Otter, Setheo et* Sacre

Les caractéristiques techniques des solveurs utilisés dans cette thèse sont résumées table 8.6. Les faiblesses de *Sacre* par rapport à ses concurrents sont premièrement son incomplétude et deuxièment l'emploi de la démodulation pour le traitement de l'égalité. La paramodulation est en effet plus puissante que la démodulation (cf. 3.5.1).

La force de *Sacre* tient à l'utilisation de techniques heuristiques puissantes qui limitent l'espace de recherche à explorer. Notamment, le mode `-auto` fait appel à des techniques de limitations comme le *weighting* ou l'option `-freeze` dont le comportement est explicité dans la section suivante. Ces techniques permettent de résoudre plus efficacement (comme le montrent les résultats récapitualitifs du tableau 8.5) un grand nombre de problèmes résolus par Otter.

8.2 Résultats suivant une approche Programmation en Logique

8.2.1 Influence des heuristiques

La table 8.7 illustre l'influence des paramètres utilisés pour la résolution de deux problèmes à caractère procédural, en l'occurrence le coloriage de cartes. Le problème *coloriel.pb* correspond au problème de coloriage présenté dans le chapitre précédent et *colorie5.pb* est un problème du même type mais un peu plus complexe.

Nous étudions pour ces problèmes, l'influence sur la résolution des options suivantes :

- `-freeze` est chargée de réduire le nombre de contrapositives résultant de la traduction du problème en CSP,

Problème	Options	Succ	Unsucc	Total	Temps	Gain %
colorie1		389	1590	1979	0.02	-
	p	324	1470	1794	0.01	9
	f	74	299	373	0.01	81
	p+f	60	270	330	0.01	83
colorie1	fch	335	1320	1655	0.01	16
	fch+p	270	1200	1470	0.01	26
	fch+f	56	210	266	0.01	87
	fch+p+f	42	180	222	0.01	89
colorie5		33022	352140	385162	0.85	-
	p	30552	329868	360420	0.78	6
	f	2635	27907	30542	0.07	92
	p+f	2400	25836	28236	0.07	93
colorie5	fch	23151	243410	266561	0.52	31
	fch+p	21564	231000	252564	0.49	34
	fch+f	2232	23462	25694	0.03	93
	fch+p+f	2064	22140	24204	0.03	94

TAB. 8.7 – *Influence des paramètres pour le coloriage de carte.*

- `-horn` réduit le nombre de contrapositives produites en se limitant à celles dont la tête correspond à un littéral positif,
- `-forward-checking` consiste à réorganiser les littéraux au sein des clauses lors de la résolution de manière à ce que les littéraux complétement instanciés soient testés en premier.

La colonne *Options* donne les options en ligne de commande utilisées pour résoudre ces problèmes :

p	`-horn`
f	`-freeze`
fch	`-forward-checking`

Les colonnes *Succ*, *Unsucc*, *Total* représentent respectivement le nombre de tentatives d'unification fructueuses, infructueuses et totales effectuées par le système. La colonne *gain* évalue le gain en pourcentages du nombre d'unifications éliminées par combinaison des différentes options par rapport à une résolution sans ces options.

Il apparaît clairement que l'influence de *freeze* est prépondérante puisqu'elle permet à elle seule de réduire de 92% la résolution du problème *colorie5* et de 81% celle de *colorie1*. Le *forward checking* posséde une influence moindre, de l'ordre de 16% pour *colorie1* et de 31% pour *colorie5*, mais qui reste néanmoins intéressante. L'influence de `-horn` est négligeable et tourne aux alentours de 10%.

La combinaison des deux techniques les plus performantes que sont `-freeze` et `-forward-checking`, apporte dans le cas du problème *colorie5* en gain de 93%.

8.2.1.1 L'option $-freeze$

Le gain obtenu par l'utilisation de l'option `-freeze` résulte de l'élimination de nombreuses contrapositives générées lors de la traduction du problème en CSP et dont la satisfaction en tant que contraintes amène à produire des valeurs redondantes. Ainsi pour le problème *colorie1.pb*, l'unique contrainte à satisfaire

```
coloriage( X1, X2, X3, X4 ) | -voisin(X1,X2) | -voisin(X1,X4),
            | -voisin(X2,X3) | -voisin(X2,X4) | -voisin(X3,X4).
```

produit les cinq contrapositives suivantes (les noms des symboles de prédicats ont été abrégés pour des raisons de lisibilité) :

$$v(x_1,x_2) \wedge v(x_1,x_4) \wedge v(x_2,x_3) \wedge v(x_2,x_4) \wedge v(x_3,x_4) \rightarrow c(x_1,x_2,x_3,x_4)$$
$$v(x_1,x_4) \wedge v(x_1,x_2) \wedge v(x_2,x_3) \wedge v(x_2,x_4) \wedge v(x_3,x_4) \rightarrow c(x_1,x_2,x_3,x_4)$$
$$v(x_2,x_3) \wedge v(x_1,x_4) \wedge v(x_1,x_2) \wedge v(x_2,x_4) \wedge v(x_3,x_4) \rightarrow c(x_1,x_2,x_3,x_4)$$
$$v(x_2,x_4) \wedge v(x_1,x_4) \wedge v(x_2,x_3) \wedge v(x_1,x_2) \wedge v(x_3,x_4) \rightarrow c(x_1,x_2,x_3,x_4)$$
$$v(x_3,x_4) \wedge v(x_1,x_4) \wedge v(x_2,x_3) \wedge v(x_2,x_4) \wedge v(x_1,x_2) \rightarrow c(x_1,x_2,x_3,x_4)$$

Ces contrapositives sont *équivalentes* en termes de solutions calculées puisque commencer à colorier la carte par x_1 revient à colorier la carte en commençant par x_2, x_3 ou x_4. En utilisant l'option `-freeze` on élimine 4 contrapositives dur les 5 générées lors de la traduction du problème.

8.2.1.2 L'option $-forward - cheking$

L'utilisation du *forward checking* est également un atout non négligeable, en particulier pour les problèmes relatifs à la Programmation en Logique. Ainsi, pour le problème de coloriage, les deux contrapositives suivantes ne possèdent pas la même efficacité (la notation adoptée et celle de *Sacre*) :

```
coloriage( X1, X2, X3, X4 ) <- -voisin(X1,X2) | -voisin(X1,X4),
          | -voisin(X2,X4) | -voisin(X2,X3) | -voisin(X3,X4).

coloriage( X1, X2, X3, X4 ) <- -voisin(X1,X2) | -voisin(X1,X4),
          | -voisin(X2,X3) | -voisin(X2,X4) | -voisin(X3,X4).
```

L'unification de `-voisin(X1,X2)` avec `voisin(rouge,vert)`, puis celle de `-voisin(X1,X4)` avec `voisin(rouge,vert)` instancie complétement le littéral `-voisin(X2,X4)` à `-voisin(vert,vert)` qui ne correspond pas à un coloriage valide. Pour la première contrapositive, l'impossibilité de colorier les zones `X2` et `X4` dans la même couleur est levée immédiatement puisque `-voisin(X2,X4)` est traitée juste après les deux premières instanciations de `-voisin(X1,X2)` et `-voisin(X1,X4)`. Par contre, pour la deuxième contrapositive, cette impossibilité ne sera mise à jour qu'après l'instanciation de `-voisin(X2,X3)`, ce qui revient à effectuer un travail inutile.

Le principe du *forward checking* pour les CSP consiste à éliminer les valeurs des domaines des variables non encore instanciées et qui ne seront pas consistantes avec l'instanciation courante. Le caractère singulier des CSP_{ext}^T ne permet pas d'envisager le *forward cheking* sous cet angle étant donné que les domaines sont infinis et non encore connus. Cependant, l'idée même du *forward cheking* qui consiste à éliminer les valeurs inconsistantes au plus tôt peut être reprise et se traduit par une réorganisation des littéraux aux sein des contrapositives.

Un autre avantage du *forward cheking* est sa compatibilité avec la déclarativité des problèmes exprimés en logique. *Prolog* rencontre également le problème que nous venons dévoquer, mais seul l'utilisateur est capable de le résoudre, en modifiant explicitement l'expression du problème. L'avantage que nous possédons par rapport à *Prolog* de ce point de vue est que le système peut se charger d'effectuer cette modification, il suffit de le lui demander.

8.3 Conclusion

L'ensemble des résultats que nous avons présentés démontre la viabilité de *Sacre* en tant que démonstrateur de théorèmes, mais également en tant que système de Programmation en Logique. D'un point de vue comportemental, *Sacre* est proche d'*Otter*, probablement en raison des techniques utilisées pour l'implantation de l'*unit-resulting resolution* et de la démodulation qui sont celles des démonstrateurs automatiques de thèorème de la famille *Otter*.

En ce qui concerne les problèmes de la compétition CADE-13, *Sacre* obtient sur le plan de l'efficacité de bien meilleurs résultats qu'*Otter* et *Setheo*. L'efficacité de *Sacre* s'explique par le fait que la résolution de ces problèmes est conduite en faisant appel à des heuristiques puissantes qui limitent l'espace de recherche sans pour autant éliminer les branches menant à une solution.

Chapitre 9

Conclusion et Perspectives

9.1 Travaux

Partant de la constatation qu'un problème de la logique des prédicats du premier ordre écrit sous forme normale conjonctive était susceptible d'être formulé sous forme de **Problème de Satisfaction de Contraintes**, l'objectif de cette thèse a consisté à proposer une méthode de résolution des problèmes de la logique suivant une approche CSP.

Nous avons effectivement montré que la transformation des problèmes de la logique en Problèmes de Satisfaction de Contraintes était naturelle. Le CSP résultant de cette transformation s'exprime sur l'ensemble des termes du problème logique initial. Les clauses sont quant à elles transformées en contraintes de cardinalité.

Nous avons clairement souligné le fait que le CSP obtenu ne pouvait être résolu suivant l'approche classique de résolution des CSP qui tend à travailler sur des domaines de taille fixes, alors que les CSP que nous sommes amenés à résoudre appellent une extension des domaines, nous obligeant ainsi à adapter le cadre classique de résolution et introduire la notion de CSP à *domaines étendus*.

La satisfaction des contraintes de cardinalité pose néanmoins des problèmes d'incomplétude et d'efficacité. Pour pallier à l'incomplétude, il faut utiliser une approche par *séparation des cas* qui augmente la combinatoire du problème. Afin d'améliorier l'efficacité de la résolution des contraintes, nous avons été forcés de faire appel à des techniques stratégiques et heuristiques issues de la Programmation par Contraintes et de la Démonstration Automatique. En particulier, la possibilité d'orienter les clauses se révèle un procédé tout à fait intéressant pour guider la recherche et diminuer de façon drastique la combinatoire de certains problèmes. Cependant, ces techniques restrictives et limitatives sont généralement incomplètes.

Un autre aspect intéressant de la méthode de résolution concerne la possibilité de calculer des modèles nous permettant ainsi d'attaquer certains problèmes suivant une approche Programmation en Logique et non plus simplement Démonstration Automatique. Nous sommes donc en mesure d'exhiber les réponses à une question posée et également les circonstances dans lesquelles une inconsistance apparaît.

Un autre point crucial est le fait que l'expression des problèmes en Programmation en Logique suivant notre approche est totalement déclarative. L'ordre d'apparition des clauses ou des littéraux au sein des clauses n'a aucune influence sur la résolution des problèmes car la transformation des clauses en contraintes de cardinalités opérée par le système de résolution permet un filtrage du problème ainsi qu'une réorganisation des littéraux au sein des clauses afin de rendre la recherche plus efficace.

L'implémentation que nous avons proposée se révèle efficace sur certains types de problèmes communément résolus par la plupart des démonstrateurs automatiques, mais montre ses limites pour des problèmes plus difficiles. Les difficultés que nous rencontrons à résoudre les problèmes de la compétition CADE-13 semblent provenir du traitement de l'égalité. En effet, la plupart de ces problèmes sont résolus par d'autres démonstrateurs en utilisant la *paramodulation* qui n'est pas implantée en *Sacre*.

9.2 Perspectives

Il nous semble que ce travail doit être poursuivi et complété dans les directions suivantes :

- le démonstrateur automatique *Sacre* est actuellement incomplet ce qui le rend incapable de traiter les problèmes spécifiant des informations incomplètes. Il paraît donc nécessaire de finaliser l'implantation du logiciel en intégrant un procédé d'*analyse de cas* qui permettrait d'étendre le panel de problèmes résolus par *Sacre*. L'implantation de la paramodulation est également nécessaire afin que la confrontation *Sacre*, Otter, Setheo se fasse sur un même pied d'égalité.
- un autre domaine d'investigation touche à l'utilisation d'un mécanisme comme celui mis en oeuvre dans l'hyper-linking qui nous semble intéressant à plusieurs égards. L'hyper-linking consiste à déterminer un sous-ensemble de clauses susceptibles de prouver l'inconsistance d'un ensemble de clauses. Ce sous-ensemble est tel que chaque littéral d'une clause s'unifie avec un littéral complémentaire d'une autre clause. Nous pensons que l'utilisation d'un tel mécanisme se révélerait utile pour orienter la recherche dans le cas d'une implantation de la méthode de résolution dirigée par les contraintes (voir ci-dessous).
- un autre axe d'investigation concerne l'implémentation. Celle-ci est en effet basée sur les domaines, en ce sens que dès qu'un domaine est étendu, l'ensemble des contraintes en rapport avec ce domaine (en fait son complémentaire) sont *réactivées* et doivent être satisfaites de nouveau. On peut imaginer une autre implantation axée sur les contraintes pour laquelle ce sont les contraintes qui dirigent la recherche. Cette implantation a cependant un défaut majeur : elle est beaucoup plus gourmande en espace mémoire que l'implantation que nous avons proposée puisqu'elle nécessite de dupliquer les valeurs produites par la satisfaction des contraintes au sein de chaque contrainte (cf. Annexe 1). Il serait toutefois intéressant de comparer les deux approches car la seconde pourraît se révéler plus efficace que la première.
- enfin, l'élargissement de *Sacre* au traitement des problèmes de la Programmation en Logique (qui fait actuellement l'objet d'un papier qui sera proposé prochainement) semble devoir focaliser notre attention dans les mois à venir. La possibilité de fournir une interprétation des termes formels et l'utilisation conjointe d'un solveur de contraintes sur le domaine fini entier nous permettrait alors de traiter beaucoup plus efficacement certains types de problèmes, comme par exemple celui de la factorielle.

Annexe A

Implantation basée sur les contraintes

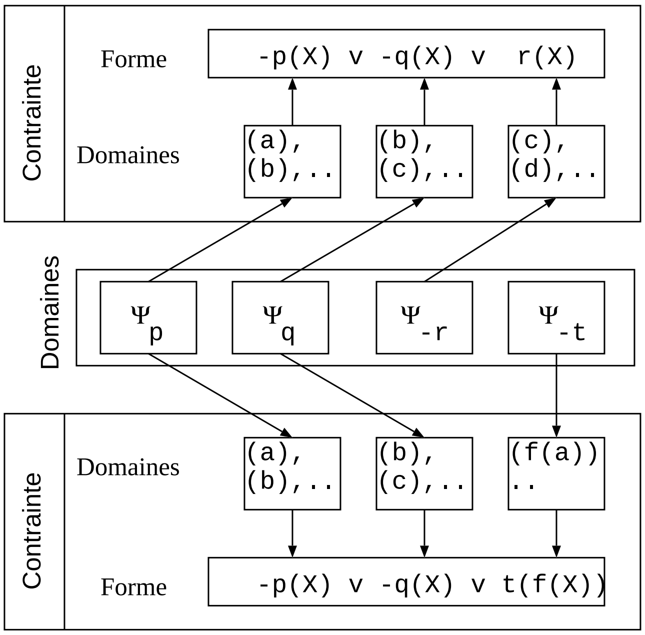

FIG. A.1 – *Implantation dirigée par les contraintes.*

Le coeur de la méthode de résolution des CSP_{ext}^T consiste schématiquement à déterminer par propagation la valeur de vérité des valeurs qui étendent les domaines.

Le système de transition basé sur la règle de *propagation* comporte une seule source de non-déterminisme : le choix de la variable à résoudre.

Résoudre une contrainte pour une variable donnée consiste à falsifier l'occurrence de la Ψ-variable associée à la variable en question et à satisfaire la contrainte résultante de manière à ce qu'une seule occurrence de Ψ-variable soit vérifiée, comme explicité dans le chapitre traitant de la méthode *Sacre*.

Chaque nouvelle valeur qui interpréte à *vrai* une variable lors de la satisfaction de la contrainte vient augmenter le domaine associé à cette variable. Elle sera ensuite réutilisée lors de futures propagations entraînant une sorte de *réaction en chaîne*.

En fait il est possible de définir un système de transition comportant non plus une mais deux sources de non-déterminisme :

- une première portant sur le choix de la variable,
- une seconde portant sur le choix de la contrainte à résoudre.

Comme on peut le constater, la méthode de résolution implantée dans *Sacre* est axée sur les variables. On peut envisager d'axer la résolution des CSP^T_{ext} sur les contraintes.

Néanmoins, le système qui en découle est nettement plus complexe à formaliser ainsi qu'à décrire. De plus il implique une redondance des valeurs à traiter (cf. figure A.1) et augmente de ce fait la complexité en espace de l'algorithme l'implantant. En revanche, il peut être doté d'un mécanisme apte à classer les valeurs à traiter et d'un mécanisme heuristique de sélection de ces valeurs beaucoup plus fin que celui utilisé dans la version actuelle de *Sacre*. Enfin, un autre atout de cette implantation est qu'elle est facilement parallélisable.

Annexe B

Grammaire de Sacre

```
SACRE        ::= INSTRUCTIONS

INSTRUCTIONS ::= INSTRUCTIONS INSTRUCTION
             | INSTRUCTION

INSTRUCTION  ::= "Clauses"     CLAUSES
             |   "Constraints" CONTRAINTES
             |   "Domaines"    DOMAINES
             |   "List"        VAR_DOMAINES
             |   "Find"

CLAUSES      | CLAUSES CLAUSE
             | CLAUSE

CLAUSE       ::= LITTERAL "<-" LITTERAUX "."
             |   LITTERAL "|"  LITTERAUX "."
             |   TETE     ":-" CORPS     "."

LITTERAUX    ::= LITTERAUX "|" LITERAL
             |   LITTERAL

TETE         ::= LITTERAL_POS

CORPS        ::= CORPS "," LITTERAL_POS
             |   LITTERAL_POS

LITTERAL_POS ::= LITTERAL

LITTERAL_NEG ::= "-" LITTERAL

LITTERAL     ::= IDF "(" TERMES ")"

TERMES       ::= TERMES "," TERME
             |   TERME

TERME        ::= IDF
             |   IDF "("  TERMES ")"
             |   "[" "]"
             |   "["  LISTE  "]"
```

```
LISTE        ::= IDF [ "," IDF ]
             |   IDF [ "|" IDF ]

DOMAINES     ::= DOMAINES DOMAIN
             |   DOMAIN

DOMAIN       ::= IDF [+|-] [ = { TERMES } ] [ "Of size" NUMBER ]

CONTRAINTES  ::= CONTRAINTES CONTRAINTE
             |   CONTRAINTE

CONTRAINTE   ::= "#(" NUMBER "," NUMBER "," LITTERAL+ ")."

VAR_DOMAINES ::= VAR_DOMAINES "," VAR_DOMAINE
             |   VAR_DOMAINE

VAR_DOMAINE  ::= IDF ["+"|"-"]
```

Annexe C

Options en ligne de commande

- **Options générales**

  `-v [n], -verbose [n]`
  Mode commentaire, `n` est un chiffre qui spécifie le niveau de détail du commentaire. (0=aucun commentaire,..., 9=commentaires détaillés).

  `-o [file], -output [file]`
  Ecriture des commentaires dans un fichier de sortie.

- **Options d'optimisation**

  `-auto`
  Mode automatique équivalent à :

  ```
  -verbose 1
  -arity 4
  -weight 5
  -deductions 1000
  -freeze
  -demodulation
  -lrpo left
  -time 300
  ```

  `-a [n], -arity [n]`
  Abandon des termes dont l'arité est supérieure ou égale au nombre `n` passé en paramètre.

  `-w [n], -weight [n]`
  Abandon des termes dont le poids est supérieur ou égal au nombre `n` passé en paramètre.

  `-c [arg], -cover [arg]`
  Cette option détermine le type de parcours lors de la recherche, le paramètre `arg` peut prendre l'une des valeurs suivantes :

`depth`	pour une reherche en profondeur d'abord
`breadth`	pour une recherche en largeur d'abord
`iterative`	pour une recherche avec approfondissements successifs

  `-d [n], -deductions [n]`
  Limitation du nombre de valeurs déduites lors de la résolution d'une contrainte au nombre `n` passé en paramètre.

  `-forward-checking`
  Réorganise les littéraux au sein des clauses pendant la résolution de manière à ce que les littéraux complètement instanciés soient testés en premier.

`-heuristic [param=arg]`
Choix d'heuristiques lors de la résolution. `param` peut prendre l'une des valeurs suivantes :

`VSH` (variable selection heuristic) représente l'heuristique de choix pour une variable, dans ce cas `arg` prend l'une des valeurs suivantes :

`MinDomain`	domaine minimal
`MaxDomain`	domaine maximal
`MinConstraint`	minimum de contraintes a satisfaire
`MaxConstraint`	maximum de contraintes a satisfaire

`CSH` (constraint selection heuristic) représente l'heuristique de choix pour une contrainte, dans ce cas `arg` prend l'une des valeurs suivantes :

`MinSize`	taille minimale
`MaxSize`	taille maximale
`MinDeduction`	minimum de deductions
`MaxDeduction`	maximum de deductions

`-s [arg], -sort [arg]`
Choix du parcours des valeurs. `arg` peut prendre l'une des valeurs suivantes :

`asc` (ascending) on va des valeurs les plus specifiques aux valeurs les plus generales,
`desc` (descending) on va des valeurs les plus generales aux valeurs les plus specifiques.

`-z, -freeze`
Interdit le traitement des contraintes redondantes

`-horn`
Traite les clauses comme des clauses de Horn. Cette option diminue le nombre de clauses à traiter.

- Options en rapport avec le traitement de l'égalité

`-demodulation`
Utiliser la démodulation pour les problèmes faisant appel à la relation d'égalité.

`-l [arg], -lrpo [arg]`
Utiliser la technique LRPO (lexical Recursive Path Ordering) pour mettre en oeuvre la démodulation. Par défaut, la démodulation de termes utilise une méthode spécifique autre que LRPO. `arg` peut prendre l'une des valeurs suivantes :
`left`
`right`
`--back_demodulation`

- **Options diverses**

`-h, -help`
Aide en ligne succinte.

`-t [n], -time [n]`
Limitation du temps de résolution à `n` secondes.

Annexe D

Java Sacre

JavaSacre est une interface graphique qui facilite l'utilisation de *Sacre*. Cette interface a été écrite en Java en utilisant la librairie Swing 1.1.3. Nous donnons dans cette annexe un exemple de session Java Sacre. L'interface est lancée par la commande `runnit`.

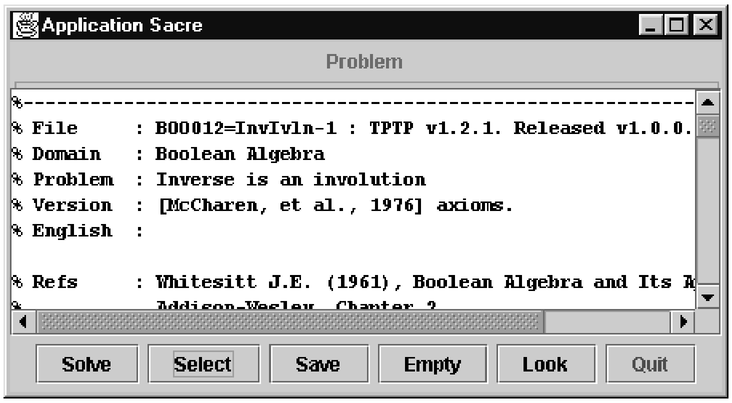

FIG. D.1 – *Fenêtre principale*

La fenêtre principale permet de choisir et éditer le problème à résoudre. Celui-ci est choisi par l'intermédiaire d'un sélecteur de fichier. On peut également créer un nouveau problème qui sera enregistré avec l'ensemble des problèmes déjà présents. Sur la figure D.1, nous avons sélectionné le problème `BOO012-1.pb`.

Pour résoudre ce problème, il faut cliquer sur le bouton `Solve` de la fenêtre principale.

Une nouvelle fenêtre s'ouvre permettant de définir les paramètres qui seront utilisés par le démonstrateur *Sacre*. On peut ainsi affiner les paramètres avant de lancer la résolution.

Sur l'exemple de la figure D.2 nous avons choisi d'utiliser les paramètres définissant le mode `-auto` de *Sacre*. Une fois les paramètres définis, on clique sur le bouton `Solve` qui ouvre une fenêtre capturant la sortie standard de *Sacre* et l'affichant dans la fenêtre courante (figure D.3).

Pour lancer la résolution, on clique sur le bouton `Run`. *Sacre* est alors exécuté avec les paramètres définis précédemment. Pour interrompre la résolution il suffit de cliquer sur le bouton `Stop` qui se substitue aux boutons `Run` et `Close` lorsque *Sacre* s'exécute. Quand la résolution arrive à son terme (ou est interrompue par l'utilisateur) on clique sur le bouton `Close` pour revenir à la fenêtre de sélection des paramètres de résolution.

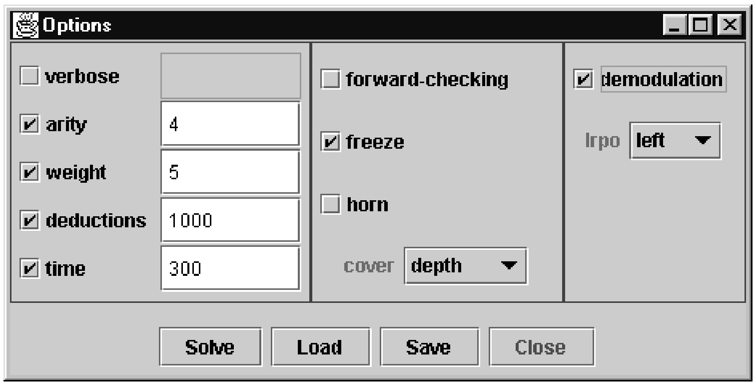

FIG. D.2 – *Fenêtre choix des paramètres de résolution*

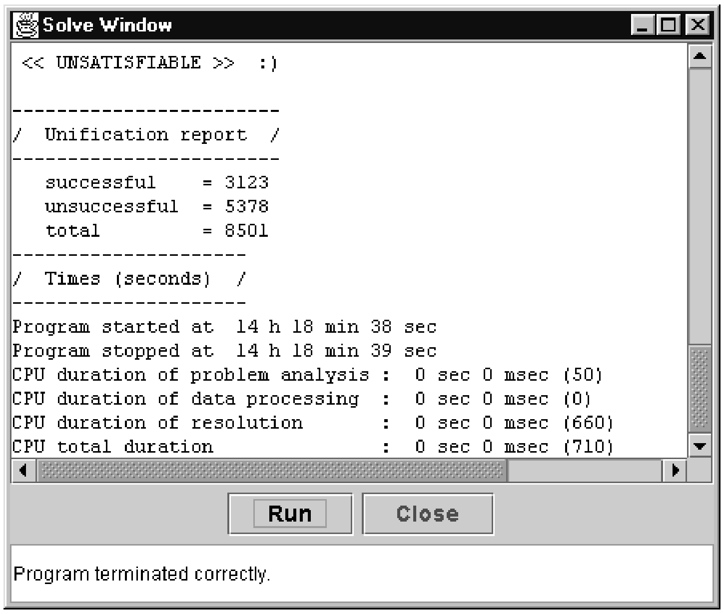

FIG. D.3 – *Fenêtre de résolution*

Annexe E

Représentation des Connaissances

Nous donnons dans cette annexe un exemple de problème concernant la représentation des connaissances en calcul propositionnel. Il s'agit du problème des logiciens de Lewis Caroll :

1. Un logicien qui dîne de deux côtelettes de porc risque de se ruiner,
2. Un joueur dont l'appétit n'est pas féroce risque de se ruiner,
3. Un homme déprimé parce qu'il s'est ruiné et qu'il risque de se ruiner à nouveau, se lève toujours à 5 heures du matin,
4. Un homme qui ne joue pas et qui ne dîne pas de deux côtelettes de porc, est assuré d'avoir un appétit féroce,
5. Un homme dynamique qui se couche avant 4 heures du matin, aurait intérêt à être conducteur de fiacre,
6. un homme doué d'un appétit féroce, qui ne s'est pas ruiné et qui ne se lève pas à 5 heures du matin, dîne toujours de deux côtelettes de porc,
7. Un logicien qui risque de se ruiner aurait intérêt à devenir conducteur de fiacre,
8. Un joueur convaincu qui est déprimé bien qu'il ne soit pas ruiné, ne court aucun risque de se ruiner,
9. Un homme qui ne joue pas, et dont l'appétit n'est pas féroce est toujours dynamique,
10. Un logicien dynamique, qui est véritablement convaincu, ne risque pas de se ruiner,
11. Un homme doué d'un appétit féroce n'a nul besoin de devenir conducteur de fiacre s'il est véritablement convaincu,
12. Un joueur qui est déprimé bien qu'il ne risque pas d'être ruiné, reste debout jusqu'à 4 heures du matin,
13. Un homme qui s'est ruiné et qui ne dîne pas de deux côtelettes de porc, aurait intérêt à devenir conducteur de fiacre à moins qu'il ne se lève à 5 heures du matin,
14. Un joueur qui se couche avant 4 heures du matin n'a nul besoin de devenir conducteur de fiacre à moins qu'il ne soit doué d'un appétit féroce.
15. Un homme doué d'un appétit féroce et qui est déprimé bien qu'il ne risque nullement de se ruiner est un joueur.

Lewis Caroll ne donne aucune indication quant à ce qui doit être recherché. On suppose que le lecteur doit déduire une relation concernant les logiciens. La transformation en contraintes de cardinalité de ce problème donne :

variable	expression
a	logicien
b	dîne de deux côtelettes de porc
c	risque de se ruiner
d	joueur
e	doué d'un appétit féroce
f	dynamique
g	s'est ruiné
h	se lève à 5 heures
i	ne se couche pas avant 4 heures
j	devrait être conducteur de fiacre
k	convaincu

$$
\begin{array}{l}
\# < 1,3,\neg a,\neg b,c > \\
\# < 1,3,\neg d,e,c > \\
\# < 1,4,f,\neg g,\neg c,h > \\
\# < 1,3,d,b,e > \\
\# < 1,3,\neg f,i,j > \\
\# < 1,4,\neg e,g,h,b > \\
\# < 1,3,\neg a,\neg c,j > \\
\# < 1,5,\neg d,\neg k,f,g,\neg c > \\
\# < 1,3,d,e,f > \\
\# < 1,4,\neg a,\neg f,\neg k,\neg c > \\
\# < 1,3,\neg e,\neg k,\neg j > \\
\# < 1,3,d,f,c,i > \\
\# < 1,4,\neg g,b,h,j > \\
\# < 1,4,\neg d,i,e,\neg j > \\
\# < 1,4,\neg e,f,c,d >
\end{array}
$$

Score parvient à exhiber 378 modèles de ce problème en 290 ms sans pour autant donner un résultat qui ait un sens. En fait l'aspect déductif est primordial pour ce problème. La production des clauses :

1. $\neg k \vee \neg a \vee i$
2. $\neg k \vee \neg a \vee h$

permet de déduire qu'*un logicien convaincu ne se couche pas avant 4h du matin et se lève à 5h.*

Bibliographie

[1] Jean-Marc ALLIOT and Thomas SCHIEX. *Intelligence Artificielle et Informatique Théorique.* Cépaduès-Éditions, 111, rue Nicolas Vauquelin, 31100 Toulouse, mars 1994.

[2] F. AMBERT, B. LEGEARD, and E. LEGROS. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *TSI*, 15(3) :2971–328, 1996.

[3] K. R. APT and M. H. VAN EMDEN. Contribution to the theory of logic programming. *Journal of the ACM*, 29 :841–862, July 1982.

[4] Olivier BAILLEUX and Jean-Jacques CHABRIER. Approximate resolution of hard numbering problems. In *In National Conference on Artificial Intelligence, AAAI'96*, pages 169–174, Portland, OR, USA, 1996.

[5] Stephan BAYERL and Reinhold LETZ. Setheo : A sequential theorem prover for first-order logic. In *Esprit'87 - Achievements and Impacts, part 1*, pages 721–735. North-Holland, 1987.

[6] Jacques BELLONE, André CHAMARD, and Annie FISCHLER. Utilisation de la programmation logique avec contraintes pour des systèmes d'aide à la décision en planification et ordonnancement chez DASSAULT AVIATION. In *IVèmes Journées Francophones de Programmation en Logique JFPL'95*, Mai 1995.

[7] B. BENHAMOU and L. SAÏS. Theoretical study of symmetries in propositionnal calculus and applications. In *Proceedings of the 11th International Conference on Automated Deduction (CADE'11)*, pages 281–294, Berlin, 1992.

[8] B. BENHAMOU and L. SAÏS. *Formules de cardinalité et Symétries en Calcul Propositionnel.* PhD thesis, Université de Provence, Marseille, 1993.

[9] B. BENHAMOU, L. SAÏS, and P. SIEGEL. Dealing with cardinality formulas in propositionnal calculus. In *Proceedings of AAAI'92, Workshop on Tractable Reasoning*, San Jose, 1992.

[10] C. BESSIÈRE and M. O. CORDIER. Arc-consistency and arc-consistency again. In *AAAI'93*, Washington, DC, USA, 1993.

[11] J. R. BITNER and E. M. REINGOLD. Backtrack programming techniques. *Communications of the ACM*, 18 :651–656, 1975.

[12] W. W. BLEDSOE. Non-resolution theorem proving. *Artificial Intelligence*, 9 :1–35, 1977.

[13] Isabelle BLOT and Jacqueline CHABRIER. Une nouvelle approche de résolution systématique sur les domaines finis entiers, basée sur la recherche locale : SCORE(FD/I). In *In CNPC'96, Conférence Nationale sur la résolution pratique des problèmes NP-Complets*, pages 119–134, Dijon, France, September 1996.

[14] Isabelle BLOT-THIBAUT. SCORE(FD/I) *: un système de Programmation par Contraintes sur les domaines finis Entiers.* PhD thesis, LIRSIA, 9, avenue Alain Savary, 21011 Dijon Cedex, France, Janvier 1998.

[15] R. S. BOYER. *Locking a restriction of resolution.* PhD thesis, University of Texas, Austin, 1971.

[16] F. BRY and A. YAHYA. Minimal model generation with positive unit hyper-resolution tableaux. In *Proceedings of the 5th Workshop on Theorem Proving with Tableaux and Related Methods*, Lectures Notes in Artificial Intelligence 1071, pages 143–159, New York, 1996. Springer-Verlag.

[17] Ralph BUTLER and Ross OVERBEEK. *A Tutorial on the Construction of High-Performance Resolution/Paramodulation Systems*, 1994.

[18] Ricardo CAFERRA and Nicolas ZABEL. A method for simultaneous search for refutation and models by equational constraint solving. *Journal of Symbolic Computation*, 13 :613–641, 1992.

[19] V. CERNY. A thermodynamical approach to the traveling salesman problem : an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1) :41–51, 1985.

[20] Jacqueline CHABRIER. Programmation par contraintes : langages, méthodes et applications sur les domaines booléens et entiers. In *Habilitation à diriger des recherches*, LIRSIA, Université de Bourgogne, Décembre 1997.

[21] Jacqueline CHABRIER, Vincent JULIARD, and Jean-Jacques CHABRIER. SCORE(FD/B) : an efficient complete local-based search method for satisfiability problems. In *CP'95 Workshop - Studying and solving hard problems*, September 1995.

[22] Jacqueline CHABRIER, Jean-Michel RICHER, and Jean-Jacques CHABRIER. Resolution of structured SAT problems with SCORE(FD/B). In *Proceedings of the CP'96 Workshop on Constraint Programming Applications*, Cambridge, Massachusetts, USA, 1996.

[23] Chin L. CHANG and Richard C. T. LEE. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., San Diego, California 92101, 1973.

[24] Heng CHU. *Semantically Guided First-Order Theorem Proving with Hyper-Linking*. PhD thesis, University of North Carolina, Chapel Hill, 1994.

[25] Heng CHU and David A. PLAISTED. CLIN-S. *Journal of Automated Reasoning*, 18(2) :183–188, 1997.

[26] Philippe CODOGNET and Daniel DIAZ. Boolean constraint solving using `clp(fd)`. In *Workshop of the International Conference on Logic Programming*, 1994.

[27] Alain COLMERAUER. Notes sur PROLOG III. In *Séminaire sur la programmation logique*, Trecastel, Mai 1986.

[28] Alain COLMERAUER. Theoretical model for PROLOG II. *Logic Programming and its applications*, pages 3–31, 1986.

[29] Alain COLMERAUER. An introduction to PROLOG II. *Communications of the ACM*, 33(7) :70–80, 1990.

[30] J. CRAWFORD, M. GINSBERG, E. LUSK, and A. ROY. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Principle and Knowledge Representation and Reasoning (KR'96)*, 1996.

[31] M. DAVIS and H. PUTNAM. A computing procedure for quantification theory. *JACM*, 7 :201–215, 1960.

[32] R. DECHTER and I. MEIRI. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of International Joint Conference of Artificial Intelligence - IJCAI'89*, pages 271–277, Detroit, 1989.

[33] J. DE KLEER and G. J. SUSSMAN. Propagation of constraints applied to circuit synthesis. *Circuit Theory and Applications*, 8 :127–144, 1980.

[34] N. DERSHOWITZ. Termination of rewriting. *Journal of Symbolic Computation*, 3 :69–116, 1987.

[35] M. DINCBAS, H. SIMONIS, and P. VAN HENTENRYCK. Solving the car sequencing problem in constraint logic programming. In *European Conference on Artificial Intelligence ECAI'88*, Munich, Germany, 1988.

[36] M. DINCBAS, P. VAN HENTENRYCK, H. SIMONIS, A. AGGOUN, T. GRAF, and F. BERTHIER. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

[37] O. DUBOIS, P. ANDRÉ, Y. BOUFKHAD, and J. CARLIER. Can a very simple algorithm be efficient for solving the sat problem? Technical report, 1993.

[38] Olivier DUBOIS. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81 :49–64, 1991.

[39] François FAGES. *Programmation Logique par Contraintes*. Ellipses, 1996.

[40] Alexandre FERNANDES DE ALENCAR and Jean-Jacques CHABRIER. Artificial intelligence applied to brazilian agro-business strategic planning and business decision support, BULL approach. In *IVèmes Journées Francophones de Programmation en Logique JFPL'95*, Mai 1995.

[41] Melvin FITTING. *First-Order Logic and Automated Theorem Proving.* Springer Verlag, 175 Fifth Avenue, New York, NY 10010, USA, 2nd edition, 1942.

[42] E. FREUDER and R. WALLACE. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[43] Dov M. GABBAY, editor. *What is a Logical System ?*, Oxford, 1994. Clarendon Press.

[44] J. GALLIER and W. SNYDER. Complete sets of transformation for general E-unification. *Theoretical Computer Science*, 67 :203–260, 1989.

[45] J. GASHING. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisfying assignment problems. In *2nd Biannual Conference of Candian Society for Computational Studies of Intelligence*, pages 268–277, Toronto, Ontario, Canada, 1977.

[46] J. GASHING. A general backtracking algorithm that eliminates most redundant tests. In *5th International Joint Conference on Artificial Intelligence*, pages 457–471, Cambridge, MA, USA, 1977.

[47] H. GEFFNER and J. PEARL. An improved constraint-propagation algorithm for diagnosis. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 1105–1111, Menlo Park, CA, USA, 1987.

[48] Tim GEISLER, Sven PANNE, and Heribert SCHÜTZ. Satchmo, the compiling and functional variants. *JAR*, 18 :227–236, 1997.

[49] G. GENTZEN. *Investigation into logical deduction*, pages 68–131. In the Collected Papers of Gerhard Gentzen, M. E. Szabo. North Holland, 1969.

[50] P. C. GILMORE. A proof method for quantification theory : its justification and realization. *IBM JRD*, pages 28–35, 1960.

[51] F. GLOVER and M. LAGUNA. *Tabu search. In Modern Heuristic Techniques for Combinatorial Problems.* McGraw-Hill, 1995.

[52] C. GOLLER, R. LETZ, K. MAYR, and J. SCHUMANN. Setheo v3.2 : Recent developments - system abstract. In *Proceedings of CADE-12*, LNAI 814, pages 778–782. Springer-Verlag, 1994.

[53] R. M. HARALICK and G. L. ELLIOT. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[54] Ryuzo HASEGAWA. Parallel theorem-proving system : Mgtp. In *Proceedings of the Fifth Generation Computer Systems FGCS'94*, Tokyo, December 1994.

[55] Jacques HERBRAND. *Recherches sur la théorie de la démonstration.* PhD thesis, Travaux de la Société des Sciences et des Lettres de varsovie, 1930.

[56] G. HUET. *Résolution d'équations dans les langages d'ordre* 1,2,...,*omega*. PhD thesis, Université d'Orsay, 1976.

[57] J. JAFFAR and S. MICHAYLOV. Methodology and implementation of a clp system. In K. Furukawa, editor, *Proceedings of the 4th ICLP, Melbourne*, 1987.

[58] Vincent JULIARD. SCORE(FD/B) *: un système de Programmation par Contraintes sur le domaine Booléen - Application.* PhD thesis, LIRSIA, 9, avenue Alain Savary, 21011 Dijon Cedex, France, Février 1998.

[59] S. KIRKPATRICK, C. D. GELATT, and M. P. VECCHI. Optimization by simulated annealing. In M. A. Fischler and O. Firschein, editors, *Readings in Computer Vision : Issues, Problems, Principles and Paradigms*, pages 606–615. Kaufmann, Los Altos, CA., 1987.

[60] D. KNUTH and P. BENDIX. Simple word problems in unversal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297, Pergamon Press, Oxford, 1970.

[61] K. E. KORF. Depth-first iterative deepening : an optimal admissible tree search. *Artificial Intelligence*, 27 :97–109, September 1985.

[62] Robert KOWALSKI. Algotihm = logic + control. *Communications of the ACM*, 22, July 1979.

[63] B. KRISNHAMURTHY. Short proofs for tricky formulas. *Acta Informatica*, pages 253–275, 1985.

[64] Jean-Louis LAURIÈRE. *Intelligence Artificielle : résolution de problèmes par l'Homme et la machine.* EYROLLES, 61, Boulevard Saint-Germain, 75005 Paris, 3rd edition, 1987.

[65] Shie Jue LEE. *CLIN : An Automated Reasoning System Using Clause Linking.* PhD thesis, University of North Carolina, Chapel Hill, 1992.

[66] B. LEGEARD and E. LEGROS. Short overview of the CLPS system. In *Proceedings of the 3rd International Symposium on programming languages implementation and logic programming PLILP'91*, pages 431–433, Passau, Germany, 26-29 August 1991, 1991.

[67] R. LETZ, K. MAYR, and C. GOLLER. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13 :297–337, 1994.

[68] John W. LLOYD. *Foundations of Logic Programming.* Springer-Verlag, Heidelberg, 2nd edition, 1987.

[69] D. W. LOVELAND. A linear format for resolution. In *Proceedings of IRIA Symposium on Automatic Demonstration*, pages 147–162, New York, 1970. Springer-Verlag.

[70] D. W. LOVELAND, David W. REED, and Debra S. WILSON. Satchmore : Satchmo with relevancy. *Journal of Automated Reasoning*, 14 :325–351, 1995.

[71] Donald W. LOVELAND. *Automated Theorem Proving : A Logical Basis.* North-Holland, New York, 1978.

[72] D. LUCKHAM. Refinements in resolution theory. In *Proceedings of IRIA Symposium on Automatic Demonstration*, pages 163–190, New York, 1970. Springer-Verlag.

[73] E. LUSK, W. MCCUNE, and R. OVERBEEK. Logic Machine Architecture : Inference mechanisms. In D. W. LOVELAND, editor, *Proceedings of the 6th Conference on Automated Deduction*, LNCS 138, pages 85–108, New York, 1982. Springer-Verlag.

[74] A. MACKWORTH. Consistency in networks of relations. *Artificial Intelligence*, 8 :99–118, 1977.

[75] A. K. MACKWORTH. On reading sketch maps. In *In Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 598–606, Menlo Park, CA, USA, 1977.

[76] R. MANTHEY and F. BRY. Satchmo : A theorem prover implemented in prolog. In *Proceedings of the 9th International Conference on Automated Deduction*, Lectures Notes in Computer Science 310, pages 415–434, New York, 1988. Springer-Verlag.

[77] A. MARTELLI and U. MONTANARI. An efficient unification algorithm. *ACM Trans. on Languages and Systems*, 4 :258–282, 1982.

[78] Philippe MATHIEU and Jean-Paul DELAHAYE. A kind of logical compilation for knowledge bases. *Theoretical Computer Science*, 131 :197–218, 1994.

[79] J. MCCHAREN, R. OVERBEEK, and L. WOS. Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications*, 2 :1–16, 1976.

[80] William MCCUNE and Larry WOS. Otter, the cade-13 competition incarnations. *JAR*, 18 :211–220, 1997.

[81] William W. MCCUNE. *Otter 3.0 Reference Manual and Guide*, 1994.

[82] N. METROPOLIS, M. N. ROSENBLUTH, A. W. ROSENBLUTH, A. H. TELLER, and E. TELLER. Equation of state calculation by fast computing machines. *Journal of Chemistry and Physics*, 21 :1087–1091, 1953.

[83] S. MINTON, M. JOHNSTON, A. PHILIPS, and P. LAIRD. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58 :161–205, 1992.

[84] U. MONTANARI. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7 :95–132, 1974.

[85] J. B. MORRIS. E-resolution : extension of resolution to include the equality relation. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'69*, pages 287–294, Washington D.C., 1969.

[86] Max MOSER and al. SETHEO and E-SETHEO, the CADE-13 systems. *JAR*, 18 :237–246, 1997.

[87] A. NEWELL, J. C. SHAW, and H. SIMON. Empirical explorations with the logic theory machine. In *Proceedings of the Western Joint Computer Conference*, pages 218–239, 1957.

[88] N. J. NILSSON. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[89] A. Oplobedu, J. Marcovich, and Y. Tourbier. Charme : un langage industriel de programmation par contraintes. In *Journées Internationales d'Avignon*, pages 55–70, 1992.

[90] Ross Overbeek. An implementation of hyper-resolution. *Computers and Mathematics with Applications*, 1 :201–214, 1975.

[91] Ross Overbeek. The cade-11 competitions : A personal view. *Journal of Automated Reasoning*, 11(3) :315–316, 1993.

[92] M. S. Paterson and M. N. Wegman. Linear unification algorithm. *Journal of Computer and Systems Science*, 16 :158–167, 1978.

[93] David A. Plaisted. A simplified problem reduction format. *Artificial Intelligence*, 18 :227–261, 1982.

[94] David A. Plaisted. Non-horn clause logic programming without contrapositives. *JAR*, 4 :287–325, 1988.

[95] D. Prawitz. *Natural Deduction. A Proof-Theoretical Study.* Acta Universitatis Stockholmiensis, Stockholm Studies in Philosophy 3, Almqvist and Wiksell, 1965.

[96] P. Prosser. Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, 9(3) :268–299, 1993.

[97] Jean-François Puget. Programmation par contraintes orientée objet. In *Journées Internationales d'Avignon*, 1992.

[98] P. W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21 :117–133, 1983.

[99] A Ramsay. Generating relevant models. *Journal of Automated Reasoning*, 7 :359–368, 1991.

[100] R. Reiter. A semantically guided deductive system for automatic theorem proving. *IEEE Transactions on Computers*, C-25(4) :328–334, 1976.

[101] Jean-Michel Richer. Symétries dans les problèmes sat structurés : Application à score booléen. *Mémoire de DEA*, 1995.

[102] Jean-Michel Richer. Systèmes formels, logique et démonstration automatique. Support de cours, 1998.

[103] J. A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1 :227–234, 1965.

[104] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12 :23–41, 1965.

[105] John A. Robinson and Larry Wos. Paramodulation and theorem proving in first-order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, 1969.

[106] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint-satisfaction problems. Technical Report ACT-AI-222-89, MCC Corp. Austin Texas, 1989.

[107] Olivier Roussel. *L'achèvement des bases de connaissances en calcul propositionnel et en calcul des prédicats.* PhD thesis, LIFL - Laboratoire d'Informatique Fondamentale de Lille, Janvier 1997.

[108] M. Rusinowitch. *Démonstration Automatique – Techniques de Réécriture.* InterEditions, 1989.

[109] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *In 10th National Conference on Artificial Intelligence*, pages 440–446, 1991.

[110] J. Slagle. Automatic theorem proving with renamable and semantic resolution. *JACM*, 14 :687–697, 1967.

[111] John Slaney. *FINDER - Finite Domain Enumerator, Version 3.0 - Notes and Guide*, 1995.

[112] John Slaney, Ewing Lusk, and William McCune. Scott : Semantically constrained otter - system description. Technical Report TR-ARP-3-94, Centre for Information Science Research, Australian National University, 1994.

[113] B. Smith. Reference manual for the environmental theorem prover : An incarnation of aura. Technical Report ANL-88-2, Argonne National Laboratory, 1988.

[114] Robert M. Smullyan. *First-Order Logic.* Springer-Verlag, Berlin, 1968.

[115] M. E. Stickel. A prolog technology theorem prover. In *International Symposium on Logic Programming*, pages 211–217, 1984.

[116] M. E. Stickel. A prolog technology theorem prover : Implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 4 :353–380, 1988.

[117] M. E. Stickel and W. M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1073–1075, Los Angeles, California, August 1985.

[118] Christian Suttner and Geoff Sutcliffe. The design of the cade-13 atp system competition. *Journal of Automated Reasoning*, 18(2) :139–162, 1997.

[119] Christian Suttner and Geoff Sutcliffe. The results of the cade-13 atp system competition. *Journal of Automated Reasoning*, 18(2) :271–286, 1997.

[120] P. Van Hentenryck and L. Michel. A modeling language for local search. In *CP'97, 3rd International Conference on Principles and Practice of Constraint Programming*, pages 745–759, Autriche, Novembre 1997.

[121] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, USA, 1989.

[122] Pascal Van Hentenryck and Yves Deville. The cardinality operator : A new logical connective for constraint logic programming. In K. Furukawa, editor, *Proceedings of the 8th ICLP, Paris, France 24-28 June 1991*, pages 745–759. The MIT Press, Cambridge, Mass., 1991.

[123] Laurent Vigneron. *Déduction Automatique avec Contraintes Symboliques dans les Théories Équationnelles.* PhD thesis, Université Henry Poincaré, Nancy I, Novembre 1994.

[124] D. L. Waltz. Generatin semantic descriptions from drawing of scenes with shadows. Technical Report AI271, M.I.T., Cambridge, MA, USA, 1972.

[125] D. L. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[126] D. H. Warren. An abstract prolog instruction set. TR 309, SRI International, October 1983.

[127] S. Winker. Generation and verification of finite models and counterexamples using an automated theorem prover answering two open questions. *J. ACM*, 29 :273–284, 1982.

[128] S. Winker, Larry Wos, and Ewing Lusk. Semigroups, antiautomorphisms and involutions : A complete computer solution to an open problem. *Math. Comp.*, 37 :533–545, 1981.

[129] WKP. Journal of automated reasoning. In *The CADE-13 Automated Theorem Proving System Competition*, volume 18. Kluwer Academic Publishers, April 1997.

[130] L. Wos, S. Winker, B. Smith, R. Veroff, and L Henschen. A new use of an automated reasoning assistant : Open questions in equivalential calculus and the study of the infinite domains. *Artificial Intelligence*, 22 :303–356, 1984.

[131] Larry Wos, Dan Carson, and George Robinson. The unit preference strategy in theorem proving. In *Proceedings of the AFIPS Conference 26*, pages 615–621, Washington D.C., 1964. Spartan Books.

[132] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning, Introduction and Applications.* McGraw-Hill, Inc., New York, 2nd edition, 1992.

[133] Larry Wos, George Robinson, and Dan Carson. Efficiency and completeness of the set of support strategy in theorem proving. *JACM*, 12, Oct. 1965.

[134] Larry Wos, George Robinson, Dan Carson, and L. Shalla. The concept of demodulation in theorem proving. *JACM*, 14 :698–704, 1967.

[135] Jian Zhang. Constructing finite algebras with falcon. *Journal of Automated Readoning*, 17 :1–22, 1996.

Résumé

La **Démonstration Automatique** s'intéresse à la preuve de théorèmes dans le cadre de la logique. Un démonstrateur automatique de théorèmes est un programme qui utilise des procédés de déduction, appelés *règles d'inférence*, pour inférer des conclusions qui découlent logiquement d'un ensemble d'axiomes. Bien que les ordinateurs soient capables de réaliser des inférences à une vitesse vertigineuse, beaucoup plus rapidement qu'aucun cerveau humain, il n'en reste pas moins que la rapidité ne représente pas toujours la panacée. La recherche de preuve en logique reste un problème combinatoire complexe car elle s'attaque à des problèmes NP-complets. La complétude des méthodes employées pour la recherche de preuve implique de parcourir l'espace de recherche dans sa totalité. On est donc parfois amené à s'intéresser à des branches de l'arbre de recherche stériles pour le but à résoudre.

La nécessité de disposer de *mécanismes de contrôle aptes à orienter la recherche* reste à l'heure actuelle une préoccupation légitime des chercheurs. Ces mécanismes sont qualifiés de *stratégies de recherche*. En conséquence, l'efficacité d'un système de démonstration automatique repose sur la conception de règles d'inférence plus performantes alliées à des stratégies de recherche plus fiables ainsi qu'à des heuristiques performantes.

La **Programmation en Logique** représente une spécialisation de la Démonstration Automatique qui privilégie l'efficacité des traitements en se restreignant à des clauses de *Horn* qui capturent le caractère procédural de certaines clauses. Un programme logique représente l'expression logique d'un algorithme qui possède l'avantage de pouvoir orienter la recherche dans une direction précise limitant ainsi l'étendue de l'espace de recherche à étudier.

Dans cette thèse, nous nous intéressons à la conception d'une nouvelle méthode de résolution des problèmes de la logique des prédicats du premier ordre exprimés sous forme normale conjonctive. L'originalité de notre approche repose sur la résolution des problèmes de la logique en tant que **Problèmes de Satisfaction de Contraintes** (CSP) adaptés au calcul des prédicats. Ces CSP peuvent être résolus comme des programmes logiques et secondés par des techniques heuristique performantes qui visent à limiter l'espace de recherche à étudier. La méthode qui résulte de nos travaux, baptisée *Sacre*, allie la déclarativité de la Démonstration Automatique à l'efficacité de la Programmation en Logique et de la Programmation par Contraintes. Elle permet de réaliser du *test de consistance* et du *calcul de modèle* sur des problèmes exprimés dans un langage de description de connaissances déclaratif et statique.

Nous avons validé notre approche en concevant un logiciel, baptisé *Sacre*, capable sous certaines conditions de rivaliser avec les démonstrateurs automatiques les plus performants à l'heure actuelle.

Mots-clés: Démonstration Automatique de Théorèmes, Problèmes de Satisfaction de Contraintes, Programmation en Logique.

Abstract

The aim of this thesis was to conceive a new efficient method for solving first-order logic problems that could serve the purpose of Automated Theorem Proving and Programming in Logic.

Our approach is firmly original in the sense that we started from the observation that logic problems expressed in conjunctive normal form could also be expressed as Constraint Satisfaction Problems (CSP). Our main work consisted in defining a method for solving the resulting CSP by integrating techniques related to the resolution of CSPs.

This method is complete and is able to check the consistency of a set of axioms (Theorem Proving) while constructing a model (Programming in Logic) over problems expressed in a purely declarative language.

The intrusion a CSP formalism inside (or into) a logical framework underlies the use of techniques that have shown their ability to solve complex combinatorial problems, especially in propositionnal calculus.

If this exercise seems simple at first, it nevertheless constitutes a stumbling block that reveals many difficulties while trying to stay as close as possible to the CSP framework. Consequently, we had to modify the original framework to be able to take into account the specificity of the CSPs.

The software *Sacre* which is the incarnation of our resolution method is able to be, under certain conditions, as efficient as the most powerful theorem provers at present time.

Keywords: Automated Theorem Proving, Constraint Satisfaction Problems, Programming in Logic, First-Order Logic.