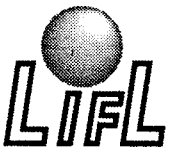


Laboratoire d'Informatique
Fondamentale de Lille

L'achèvement des bases de connaissances en calcul propositionnel et en calcul des prédicats

THÈSE

présentée et soutenue publiquement le 7 janvier 1997

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(Spécialité Informatique)

par

Olivier ROUSSEL

Composition du jury

Président :	Pierre SIEGEL
Rapporteurs :	Marie-Catherine VILAREM
	Antoine RAUZY
Examinateurs :	Jean-Paul DELAHAYE
	Gérard FERRAND
	Éric GRÉGOIRE
	Pierre MARQUIS
	Philippe MATHIEU

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - URA 369 CNRS - Bât. M3 - UFR IEEA - 59655 VILLENEUVE D'ASCQ CEDEX
Tél: (33) 20 43 44 92 - Fax: (33) 20 43 65 66 - E_mail: direction@lifl.fr

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Remerciements

La soutenance d'une thèse est un événement somme toute assez bref. Il suffit finalement d'une heure d'exposé et de quelques dizaines de minutes de délibération pour que la page soit tournée. Mais cette brièveté cache une genèse plus tourmentée. Derrière les quelques secondes qui suffisent au jury pour rendre ses conclusions, il y a le temps passé par chacun des examinateurs à lire un mémoire malheureusement toujours trop austère. Merci à Gérard Ferrand, Eric Grégoire, Pierre Marquis et Pierre Siegel d'avoir pris ce temps et de m'avoir fait l'honneur de participer à ce jury. Il y a également la lecture plus attentive encore des rapporteurs qui se sont penchés sur tous les détails de ce mémoire. Je remercie vivement Marie-Catherine Vilarem et Antoine Rauzy de la confiance qu'ils m'ont témoignée en acceptant cette tâche. Merci tout particulièrement à Pierre Marquis et Antoine Rauzy pour les conseils qu'il m'ont prodigués.

Mais avant tout, la soutenance d'une thèse marque l'aboutissement d'une période de formation à la recherche et par la recherche. Cette formation a été initiée par Philippe Mathieu et Jean-Paul Delahaye. Leur encadrement depuis le début de mon DEA a été précieux et plus particulièrement les longues heures de discussions passées avec Philippe. Merci à lui pour les relectures des versions préliminaires de ce mémoire et pour les conseils qu'il a su me prodiguer tout au long de ces années. Merci également à Jean-Marc Talbot pour ses pertinentes remarques.

Merci aussi au conseil régional Nord-Pas de Calais et au CNRS qui ont rendu possible ce travail de longue haleine en acceptant de le financer.

Je tiens également à remercier tous ceux qui m'ont permis d'accéder en thèse. La liste des enseignants qui sont responsables de ma réussite est trop longue pour être énumérée ici. Je ne retiendrai que trois noms, Alain Loncke, Jean-Claude Vieillard et Alain Allouchery pour représenter chacun de ceux qui, de la maternelle au DEA, ont ouvert mon esprit. Merci tout particulièrement à ces trois professeurs pour le petit plus indéfinissable qu'ils m'ont apporté.

Enfin, le plus grand mérite revient sans-doute à mes parents qui, tout au long de ce quart de siècle, ont toujours su m'encourager et me donner les moyens de cette réussite. Merci surtout à eux pour m'avoir donner le goût d'apprendre. Sans eux, cette thèse n'existerait pas.

À mes parents, à qui je dois tant.

Table des matières

Liste des figures

Introduction

L'un des problèmes centraux de l'intelligence artificielle consiste à programmer la machine pour qu'elle effectue des inférences proches du raisonnement humain et ce, le plus rapidement possible. Il existe dans ce but de multiples formalismes qui tentent de capturer l'essence de l'intelligence. Chacun possède ses forces et ses faiblesses.

Dans cette thèse, nous nous intéressons plus particulièrement à l'un de ces formalismes parce qu'il nous apparaît assez proche du raisonnement humain et qu'il est en même temps très efficace. Il s'agit du modus ponens dans le cadre de la logique booléenne classique. Cette règle d'inférence qui est le fondement du chaînage avant est sans doute la plus simple des règles de déduction. De plus, la logique booléenne semble la plus naturelle. Cependant, si ce formalisme présente des avantages, il a aussi une faiblesse de taille : il ne garantit pas d'effectuer toutes les déductions que permet la logique booléenne habituelle (ce qui le rapproche également du raisonnement humain).

L'objectif de cette thèse est d'étendre les précédents travaux sur l'achèvement [Mat91] aussi bien dans le cadre du calcul propositionnel que dans celui du calcul des prédicats. L'achèvement est une méthode qui pallie de manière originale le défaut d'incomplétude du chaînage avant en ajoutant les règles qui codent les déductions que l'algorithme ne sait effectuer. Si l'on considère par exemple la connaissance ci-dessous

S'il ne pleut pas et qu'il ne neige pas, alors il fait beau.
Si je fais du vélo, il ne pleut pas.
Si je fais du vélo, il ne neige pas.

le chaînage avant sait déduire qu'il fait beau quand je fais du vélo, mais ne sait pas conclure que je ne fais pas de vélo quand il ne fait pas beau. L'achèvement de cette connaissance consiste donc à rajouter

S'il ne fait pas beau, alors, je ne fais pas de vélo

L'ajout de cette simple règle permet maintenant au chaînage avant d'effectuer toutes les déductions possibles. Il s'agit d'une méthode de compilation puisque le chaînage avant sur la base achevée sera complet quelle que soit la base de faits utilisée par la suite. Cette transformation est également une compilation *logique* puisque le résultat de l'achèvement est une base exprimée en logique pure. Ce type de méthode permet de conserver les avantages du modus ponens qui sont sa simplicité et son efficacité tout en éliminant son défaut d'incomplétude.

Les résultats que nous avons obtenus sont multiples. En calcul propositionnel tout d'abord, et suite aux travaux de [MD94a], nous avons obtenu un algorithme efficace d'achèvement par parties. Cet algorithme permet de décomposer une base en recherchant les composantes 2-connexes d'un graphe qui lui est associé, ce qui permet de réduire considérablement la complexité de l'achèvement. En effet, dans les cas favorables, il est possible de passer d'une complexité exponentielle en temps à une complexité polynomiale. Les résultats obtenus

pour l'achèvement par parties ont été raffinés dans l'achèvement par cycles. Cette méthode consiste à identifier de manière fine les clauses nécessaires à l'achèvement en recherchant certains cycles d'un graphe associé à la base. Elle repose sur l'identification d'une condition nécessaire et suffisante d'achèvement basée sur la notion de résolution avec fusion. Cette méthode est particulièrement intéressante car elle permet de se concentrer sur les règles qu'il faut ajouter, par opposition aux méthodes qui produisent un grand nombre de règles dont seules certaines sont indispensables à l'achèvement. Par ailleurs, cette méthode est intrinsèquement incrémentale et graphique. Elle convient donc tout particulièrement pour un usage interactif de l'achèvement tel que la construction d'un éditeur de règles. Ce dernier permet d'identifier à chaque ajout de connaissance les déductions que le chaînage avant ne saura effectuer et d'en expliquer graphiquement la cause à l'utilisateur. Cette explication est par ailleurs très intuitive. Nous avons également montré expérimentalement que l'usage de l'achèvement permet d'effectuer des inférences complètes plus rapidement qu'avec une méthode basée sur l'usage d'un test complet de satisfiabilité, ce qui justifie l'emploi du terme «compilation».

L'autre résultat essentiel obtenu est l'extension de la notion d'achèvement aux bases du calcul des prédicats composées de clauses quantifiées universellement, cadre qui étend les langages DATALOG et Prolog. Ce résultat ne découle pas trivialement de ceux obtenus en calcul propositionnel. On peut en effet montrer que conserver les mêmes concepts interdit d'obtenir dans tous les cas une base achevée finie. Une solution possible pour éviter ce problème consiste à étendre le vocabulaire de la base initiale. Cette extension ne remet pas en cause l'usage du chaînage avant (il suffit de filtrer les faits produits) et évite de devoir prévoir à la compilation toutes les exécutions possibles de l'algorithme. Nous avons d'abord montré que cette extension permet effectivement de réaliser un achèvement total en utilisant la notion de méta-interprète. Nous avons également établi une méthode basée sur des considérations pratiques qui permet d'éviter l'extension du vocabulaire mais ne garantit qu'un achèvement partiel. Lorsqu'un achèvement total est nécessaire, la méthode à employer est une extension de l'achèvement par cycle au premier ordre. Cette méthode hérite de certains des avantages de la recherche de cycles en calcul propositionnels. Mais surtout, elle permet d'expliquer simplement pourquoi la base achevée peut être infinie et comment l'extension du vocabulaire permet de contourner ce problème.

La première partie de ce mémoire présente la problématique de l'achèvement. Le premier chapitre présente les notions classiques du calcul propositionnel. Le deuxième chapitre détaille le fonctionnement du chaînage avant, aussi bien sur des règles que sur des clauses. La notion d'achèvement est alors définie dans le chapitre trois. Cette partie se conclut sur la présentation d'autres approches de la compilation logique (chapitre 4).

La deuxième partie de ce document détaille les méthodes d'achèvement en calcul propositionnel. Le chapitre 5 décrit les méthodes sémantiques. La méthode d'achèvement par parties est présentée au chapitre 6. Le chapitre suivant traite des méthodes syntaxiques et plus particulièrement de l'achèvement par cycles. Une comparaison entre l'usage de l'achèvement et celui d'une méthode complète d'inférence clôt alors cette étude du cas propositionnel.

La troisième et dernière partie de cette thèse est consacrée à l'étude du calcul des prédicats. Le chapitre 9 rappelle les notions indispensables. Le chapitre 10 présente le fonctionnement du chaînage avant au premier ordre. Les problèmes posés par l'achèvement dans ce formalisme sont alors détaillés dans le chapitre suivant. Le chapitre 12 étend la condition nécessaire et suffisante d'achèvement au premier ordre. Une méthode d'achèvement par méta-interprète est alors exposée dans le chapitre 13. D'un intérêt plus pratique, une méthode utilisant les champs de production fait l'objet du chapitre suivant. Enfin, le chapitre 15 étudie comment l'achèvement par cycles peut être étendu au premier ordre et en quoi il permet d'expliquer et de résoudre certains des problèmes de l'achèvement pour le calcul

des prédicats.

Première partie

Problématique

Chapitre 1

Définitions et notations

Nous rappelons ici brièvement les notions de logique propositionnelle ainsi que les notations que nous utilisons. Des présentations plus complètes peuvent être trouvées dans [CL73].

La logique propositionnelle s'intéresse à la valeur de vérité des propositions. Une proposition est soit vraie, soit fausse. On parle alors de logique bivaluée par opposition à des logiques où les propositions peuvent avoir d'autres statuts. Une proposition atomique est une proposition qui ne peut être davantage décomposée. Une proposition non atomique est formée de connecteurs logiques reliant d'autres propositions.

Définition 1:

Une **proposition atomique** aussi appelée **atome** est une variable booléenne. Elle peut donc prendre pour **valeur de vérité** $\mathbb{V}$ ou $\mathbb{F}$.

Une **proposition** ou encore **formule** du calcul propositionnel est

- soit un atome
- soit la **négation** d'une formule $(\neg f)$
- soit la **conjonction** de deux formules $(f_1 \wedge f_2)$
- soit la **disjonction** de deux formules $(f_1 \vee f_2)$
- soit l'**implication** de deux formules $(f_1 \rightarrow f_2)$
- soit l'**équivalence** de deux formules $(f_1 \leftrightarrow f_2)$

Une formule doit être de taille finie.

Un **littéral** est un atome (a) (**littéral positif**) ou la négation d'un atome $(\neg a)$ (**littéral négatif**).

Le **vocabulaire** d'une formule est l'ensemble des symboles qui la compose. Formellement, il est représenté par le couple $(\mathcal{Q}, \mathcal{A})$ de l'ensemble des connecteurs et de l'ensemble des atomes qui y apparaissent. Cependant, il est assez inutile et pesant d'inclure dans le vocabulaire les connecteurs logiques en calcul propositionnel. On note donc $\boldsymbol{Voc(f)}$ l'ensemble des atomes qui figurent dans la formule f. On note également $\boldsymbol{Lit(f)}$ l'ensemble des littéraux qui peuvent être construits à partir des atomes de $Voc(f)$.

Note 2:

On convient d'une priorité entre les connecteurs de manière à éviter de trop nombreuses

parenthèses. Le connecteur le plus prioritaire est $\neg$ suivi de $\vee$ puis de $\wedge$ et enfin de $\rightarrow$ et $\leftrightarrow$.

La valeur de vérité de chaque formule se déduit de la valeur de vérité des éléments de son vocabulaire.

Définition 3:

Une **interprétation** I en calcul propositionnel est une application de l'ensemble des formules dans $\{\mathbb{V}, \mathbb{F}\}$ qui est définie par la valeur de vérité ($\mathbb{V}$ ou $\mathbb{F}$) de chaque atome.

L'interprétation $I(f)$ d'une formule se calcule suivant les règles ci-dessous :

- $I(\neg f) = \mathbb{V} \Leftrightarrow I(f) = \mathbb{F}$
- $I(f_1 \wedge f_2) = \mathbb{V} \Leftrightarrow I(f_1) = I(f_2) = \mathbb{V}$
- $I(f_1 \vee f_2) = \mathbb{F} \Leftrightarrow I(f_1) = I(f_2) = \mathbb{F}$
- $I(f_1 \rightarrow f_2) = \mathbb{F} \Leftrightarrow I(f_1) = \mathbb{F} \wedge I(f_2) = \mathbb{V}$
- $I(f_1 \leftrightarrow f_2) = \mathbb{V} \Leftrightarrow I(f_1) = I(f_2)$

Toute formule qui n'est pas interprétée à $\mathbb{V}$ est interprétée à $\mathbb{F}$ et réciproquement.

Une **interprétation partielle** est une fonction qui n'est pas définie pour tous les atomes (ni donc pour toutes les formules).

Une **interprétation totale** est une fonction qui est définie pour tout atome et donc pour toute formule.

Définition 4:

Une formule f est **satisfaite** par une interprétation I si et seulement si $I(f) = \mathbb{V}$. Une formule **valide** ou **tautologie** est une formule satisfaite par toutes les interprétations. Une formule est **insatisfiable**) si aucune interprétation ne la satisfait. Elle est **satisfiable** s'il existe une interprétation qui la satisfait. Lorsque l'on utilise des méthodes syntaxiques telles que la résolution pour décider de la satisfiabilité d'une formule, on parle de formule **consistante** lorsqu'elle est satisfiable et de formule **inconsistante** lorsqu'elle ne l'est pas.

Un **modèle total** d'une formule est une interprétation totale qui la satisfait. Un **modèle partiel** d'une formule est une interprétation partielle I telle que tout prolongement de I en une interprétation totale est une interprétation qui satisfait la formule. Un **modèle** d'une formule est un modèle partiel ou total.

On définit une notion d'implication et d'équivalence entre formules comme suit.

Définition 5:

Il y a **implication sémantique** entre une formule f_1 et une formule f_2 si et seulement si tout modèle de f_1 est un modèle de f_2. On note $\boldsymbol{f_1 \models f_2}$ (f_1 implique f_2).

Deux formules f_1, f_2 sont **sémantiquement équivalentes** (noté $\boldsymbol{f_1 \equiv f_2}$) si et seulement si $f_1 \models f_2$ et $f_2 \models f_1$.

On remarquera en particulier pour la suite que $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$. Par ailleurs, les opérateurs $\vee$ et $\wedge$ sont associatifs et commutatifs pour cette relation d'équivalence. Le $\wedge$ se distribue sur le $\vee$ de même que le $\vee$ sur le $\wedge$. Il y a également dualité entre ces connecteurs ($\neg(f_1 \vee f_2) \equiv \neg f_1 \wedge \neg f_2$ et $\neg(f_1 \wedge f_2) \equiv \neg f_1 \vee \neg f_2$).

Muni de cette relation d'équivalence, il est possible de réécrire une formule pour aboutir à une forme normale.

Définition 6:

Une formule est en **forme normale conjonctive (CNF)** s'il s'agit d'une conjonction de disjonctions de littéraux.

Une formule est en **forme normale disjonctive (DNF)** s'il s'agit d'une disjonction de conjonctions de littéraux.

Définition 7:

Une **clause** est un ensemble de littéraux que l'on assimile à la disjonction de ces littéraux.

La **clause vide** est un ensemble vide de littéraux. Elle est notée $\square$ et est insatisfiable.

Une **clause de Horn** est une clause contenant au plus un littéral positif.

On note $\boldsymbol{length(C)}$ le nombre de littéraux contenus dans la clause C.

Définition 8:

Une **règle** est une formule de la forme $l_1 \wedge l_2 \wedge \ldots \wedge l_{n-1} \rightarrow l_n$ où chaque l_i est un littéral. l_n est appelé **conclusion** de la règle tandis que $l_1 \wedge l_2 \wedge \ldots \wedge l_{n-1}$ est appelé **prémisse** ou **condition** de la règle. On peut aussi noter une règle sous la forme *si* l_1 *et* l_2 *et* ... *et* l_{n-1} *alors* l_n. Cette formule est sémantiquement équivalente à la clause $\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_{n-1} \vee l_n$.

Définition 9:

Une **base** est un ensemble fini de formules que l'on assimile à la conjonction de ces formules.

Définition 10:

Une clause C_1 **subsume** au sens large une clause C_2 si $C_1 \models C_2$. Il y a subsomption au sens strict si $C_2 \not\models C_1$. En calcul propositionnel, $(C_1 \models C_2) \Leftrightarrow (C_2$ est une tautologie$) \vee (C_1 \subseteq C_2)$ (inclusion ensembliste large).

Une conjonction de littéraux M_1 **subsume** au sens large une conjonction de littéraux M_2 si $M_2 \models M_1$. Il y a subsomption au sens strict si $M_1 \not\models M_2$. En calcul propositionnel, $(M_2 \models M_1) \Leftrightarrow (M_1 \subseteq M_2)$

Définition 11:

Un **impliqué** d'une base $\mathcal{B}$ est une clause C telle que $\mathcal{B} \models C$.

Un **impliqué premier** d'une base est un impliqué qui n'est strictement subsumé par aucun autre.

On note $\boldsymbol{PI(\mathcal{B})}$ l'ensemble des impliqués premiers de la base $\mathcal{B}$.

Un **impliquant** d'une base $\mathcal{B}$ est une conjonction de littéraux M telle que $M \models \mathcal{B}$.

Un **impliquant premier** d'une base est un impliquant qui n'est strictement subsumé par aucun autre.

En calcul propositionnel, les notions d'impliquant et de modèle partiel se confondent.

Définition 12:

On appelle **couverture d'impliquants** (ou **couverture de modèles**) d'une base $\mathcal{B}$ un ensemble d'impliquants de $\mathcal{B}$ qui, vu comme la disjonction de ces impliquants, lui est sémantiquement équivalent.

Similairement, une **couverture d'impliqués** d'une base $\mathcal{B}$ est un ensemble d'impliqués de $\mathcal{B}$ qui, vu comme une conjonction de ces impliqués, lui est sémantiquement équivalent.

Une couverture $\mathcal{C}$ d'impliquants ou d'impliqués d'une base $\mathcal{B}$ est **irredondante** si $\nexists I \in \mathcal{C}$ tel que $\mathcal{C} - I$ soit également une couverture de $\mathcal{B}$

Chapitre 2

Le chaînage avant en calcul propositionnel

Ce chapitre présente le fonctionnement du chaînage avant dans le cadre du calcul propositionnel, aussi bien sur des règles que sur des clauses. Nous détaillons ensuite les points forts et les défauts de cet algorithme.

Le chaînage avant a pour but de produire les littéraux impliqués par la connaissance et ce, par usage répété du modus ponens (calcul d'un point fixe). On considère que la connaissance est représentée par deux composantes

- un ensemble $\mathcal{B}$ de règles ou de clauses (**base de règles/clauses**)
  Il représente la connaissance du monde qui nous permet d'effectuer une inférence.
- un ensemble F de littéraux appelés **faits** (**base de faits**)
  Les faits représentent ce que l'on a appris du monde ou déduit sur ce dernier.

Le chaînage avant calcule une fonction qui à un couple $(\mathcal{B}, F)$ associe un ensemble de littéraux impliqués par $\mathcal{B} \cup F$.

Notation

On note par $\boldsymbol{Fwch(\mathcal{B} \cup F)}$ l'ensemble des littéraux produits par chaînage avant à partir de la base de clauses ou de règles $\mathcal{B}$ et de la base de faits F.

Le chaînage avant est donc un algorithme de production. Il se distingue fondamentalement des algorithmes d'interrogation qui, à un triplet (base, faits, littéral) $(\mathcal{B}, F, L)$ associent la réponse *oui* ou *non* selon que le littéral L est impliqué par la connaissance $\mathcal{B} \cup F$ ou pas. Un algorithme de production peut toujours être utilisé pour répondre à une interrogation. Il suffit d'ajouter les faits $F \cup \neg L$ et voir si cela produit une inconsistance. Réciproquement, un algorithme d'interrogation peut être utilisé pour créer un algorithme de production comme nous le verrons dans le chapitre 8. Cependant, il s'agit toujours d'une simulation, et l'efficacité obtenue n'est pas la même que celle de l'algorithme original.

En fait, un algorithme d'interrogation est adaptée au cas où l'on connaît ce que l'on veut prouver. L'algorithme de production quant à lui est conçu pour des situations où l'on ne sait pas précisément ce que l'on va prouver. Ce type de situation se rencontre fréquemment. On peut par exemple prendre le cas d'un raisonnement médical. L'ensemble des faits représente les symptômes que présente le patient. La base de règles code la connaissance du médecin et permet de produire un diagnostic. Naturellement, on ne sait pas à l'avance de quelle maladie il s'agit, ni même s'il n'y en a qu'une ! Dans cette situation, un algorithme de

production sera le plus adapté puisqu'il saura utiliser simultanément tous les symptômes pour effectuer son raisonnement tandis qu'un algorithme d'interrogation obligerait à passer en revue tous les diagnostics possibles.

Nous détaillons maintenant comment l'algorithme du chaînage avant fonctionne sur des règles ou des clauses.

2.1 Sur des règles

En calcul propositionnel, l'algorithme du chaînage avant fonctionnant sur une base de règles et un ensemble de faits est le suivant :

1: **function** RulesFwch($\mathcal{B}$: base de règles, F base de faits) :base de faits
2: **repeat**
3: **if** $\exists$ une règle $l_1, l_2, \ldots, l_n \rightarrow l$ de $\mathcal{B}$ telle que chaque l_i figure dans F **then**
4: **if** l n'est pas déjà dans F **then**
5: ajouter l à F
6: **end if**
7: **end if**
8: **until** une contradiction est détectée ou aucun fait n'a été ajouté
9: **return** F
10: **end function**

Il s'agit d'une simple répétition du modus-ponens menée jusqu'a ce qu'on atteigne un point fixe.

2.2 Sur des clauses

On peut définir un autre algorithme qui fonctionne cette fois sur une base de clauses et un ensemble de faits. Nous continuons à l'appeler chaînage avant car nous verrons que, abstraction faite de certains détails syntaxiques, il effectue le même calcul que le chaînage avant habituel sur des règles.

1: **function** ClausesFwch($\mathcal{B}$: base de clauses, F base de faits) :base de faits
2: **repeat**
3: **if** $\exists$ une clause $l_1 \vee l_2 \vee \ldots \vee l_n \vee l$ de $\mathcal{B}$ telle que $\forall i$, $\neg l_i \in F$ **then**
4: **if** l n'est pas déjà dans F **then**
5: ajouter l à F
6: **end if**
7: **end if**
8: **until** une contradiction est détectée ou aucun fait n'a été ajouté
9: **return** F
10: **end function**

Cet algorithme est aussi connu dans la littérature sous d'autres noms. Il est en particulier très proche de l'*unit-resolution* [CL73]. Il s'en démarque cependant puisque si l'unit-resolution permet la production de clauses de longueur quelconques, cet algorithme ne produit que des clauses de longueur au plus une (les faits). Pour cette raison, on parle d'*unit-propagation* de littéraux. On retrouve également le même algorithme dans [McA90][FdK93] ou [Dal95] sous le nom de BCP (Boolean Constraint Propagation). Par ailleurs, on le retrouve aussi dans de nombreux tests de satisfiabilité. Par exemple, dans [DABC94] la règle *unit-clause* utilisée dans la recherche locale de C-SAT est en fait un chaînage avant. Il en va de même pour le Forward Checking utilisé dans les systèmes de résolution de contraintes.

Nous n'avons pas pour notre part de réelle préférence sur le nom à donner à cet algorithme. En revanche, il est à notre avis important de se rendre compte qu'il s'agit d'une autre formulation du chaînage avant, comme le montre le paragraphe suivant.

2.3 Relation entre les deux formes de chaînage avant

À première vue, le chaînage avant sur des clauses est plus puissant que celui sur des règles. En effet, considérons la règle $a \rightarrow b$ et sa traduction clausale $\neg a \vee b$. Il apparaît immédiatement qu'à partir de $\neg b$ on saura prouver par chaînage avant sur la clause que $\neg a$ est impliqué. En revanche, la règle restera inutilisable puisqu'il faudrait produire sa prémisse. Mais en réalité, la différence entre les deux algorithmes est purement syntaxique. En fait, cet exemple est biaisé car la clause $\neg a \vee b$ représente deux règles : $a \rightarrow b$ et sa contraposée $\neg b \rightarrow \neg a$. Lorsque la clause contient plus de deux littéraux, il faut utiliser la notion de variante [Mat91] pour généraliser celle de contraposée.

Définition 13:

Une **variante** d'une clause C est une règle notée $\boldsymbol{Var(C)}$ qui a C pour forme clausale.

Une **variante** d'une règle R est une règle $Var(R)$ qui a même forme clausale.

Une clause a autant de variantes qu'elle contient de littéraux.

Exemple 14:

La clause $a \vee b \vee c$ possède trois variantes

- $\neg b \wedge \neg c \rightarrow a$
- $\neg c \wedge \neg a \rightarrow b$
- $\neg a \wedge \neg b \rightarrow c$

Une variante permet en fait d'orienter une clause, de préciser dans quel sens on compte l'utiliser. La différence entre les deux types de chaînage avant réside donc dans un simple marquage syntaxique, comme le montre la proposition suivante.

Proposition 15:

Soit $\mathcal{B}$ une base de clauses. Pour tout ensemble F de faits, le chaînage avant sur les clauses de $\mathcal{B}$ produit les mêmes littéraux que le chaînage avant sur les règles de $Var(\mathcal{B})$.

Soit $\mathcal{B}$ une base de règles. Pour tout ensemble F de faits, le chaînage avant sur les règles de $Var(\mathcal{B})$ produit les mêmes littéraux que le chaînage avant sur la traduction clausale de $\mathcal{B}$.

Compte tenu de ce résultat, nous travaillerons aussi bien sur des clauses que sur des règles, d'autant plus que le calcul des variantes est une opération très simple de complexité linéaire en temps et en espace par rapport à la taille de la base.

2.4 Forces et faiblesse

Le chaînage avant présente à notre avis des intérêts multiples. C'est d'abord un algorithme d'une simplicité déconcertante. De ce fait, l'utilisateur n'a guère de peine à en comprendre et maîtriser le fonctionnement. C'est pourquoi on le retrouve aussi bien dans les applications pratiques de type systèmes experts, que dans des utilisations plus théoriques en sémantique des langages logiques sous le nom d'opérateur conséquence immédiate. Il est de plus très efficace puisqu'il est de complexité linéaire par rapport à la taille de la base. D'autre part, sa simplicité le rend très facile à implanter, aussi bien en environnement centralisé que réparti. Il est également très facile de le compiler à un niveau très bas (en assembleur par exemple) pour en accélérer l'exécution. On peut même imaginer l'implanter sans peine au niveau du silicium dans des circuits spécialisés ! Mais surtout, l'intérêt principal du chaînage avant est d'être un algorithme de production qui, idéalement, produit tous les littéraux impliqués par la connaissance.

Malheureusement, et c'est peut-être là son seul défaut, le chaînage avant n'est pas complet pour la logique booléenne classique. Par exemple, à partir de la base de clauses $\{\neg a \vee b, \neg a \vee c, \neg b \vee \neg c \vee d\}$ et du fait $\neg d$, il est impossible de dériver $\neg a$ par chaînage avant alors qu'il est conséquence logique dans le cadre booléen. En fait, le chaînage avant est complet pour une logique trivaluée [Del87]. Cependant, cette logique ne permet pas toutes les déductions qui découlent du bon sens. Une des solutions possibles à cette incomplétude est l'achèvement.

Chapitre 3

L'achèvement

L'idée maîtresse de l'achèvement est simple et est due à Philippe Mathieu. Elle est développée entre autres dans [MD90b], [Mat91] et [MD94b]. Puisque le chaînage avant ne sait pas effectuer toutes les déductions qu'on attend de lui, pourquoi ne pas les lui apprendre ? En effet, il suffit d'ajouter à la base les règles qui représentent les déductions que l'algorithme ne sait pas effectuer pour que le calcul devienne complet.

Achever une base consiste donc à ajouter des règles ou des clauses pour que le chaînage avant devienne complet, sans toutefois modifier la sémantique de la base. Bien sûr, on désire ajouter le moins de clauses possible le plus rapidement possible. Mais surtout, on souhaite ne pas devoir modifier la base lorsqu'on change les faits. De la sorte, achever une base devient une compilation dont le résultat sera utilisable pour toute base de faits à venir, ce qui permettra de rentabiliser le coût de l'achèvement.

Selon que l'on dispose ou non d'informations sur les bases de faits qui seront utilisées par la suite, on distingue deux types d'achèvement : l'achèvement total ou partiel.

3.1 Achèvement total

Si l'on ne dispose pas d'information sur les bases de faits que l'on compte utiliser, il est nécessaire d'ajouter toutes les déductions que le chaînage avant ne sait pas effectuer. On parle alors d'achèvement total car il garantit la complétude pour toute base de faits.

Définition 16:

On appelle **achèvement total** l'opération qui, à une base $\mathcal{B}$ de formules, associe une base (de règles ou de clauses selon le type de chaînage avant utilisé) notée $\boldsymbol{Achvt(\mathcal{B})}$ telle que

- $\mathcal{B} \equiv Achvt(\mathcal{B})$ (équivalence sémantique)
- $\forall F$ base de faits, $\forall l$ littéral, $\mathcal{B} \cup F \models l \Leftrightarrow l \in Fwch(\mathcal{B} \cup F)$
- $\forall F$ base de faits, $\mathcal{B} \cup F \models \square \Leftrightarrow \exists l,\ \{l, \neg l\} \subseteq Fwch(\mathcal{B} \cup F)$

On impose que la base achevée soit sémantiquement équivalente à la base initiale car on ne souhaite pas dénaturer la connaissance. Par ailleurs, si $\mathcal{B} \cup F$ est insatisfiable, l'inconsistance doit être détectée par le chaînage avant.

Exemple 17:

La base $\mathcal{B}$ ci-dessous

$$\begin{cases} a \vee b \\ a \vee c \\ \neg b \vee \neg c \vee d \end{cases}$$

n'est pas totalement achevée puisque $a \notin Fwch(\mathcal{B} \cup \{\neg d\})$ alors que $\mathcal{B} \cup \{\neg d\} \models a$.

Il suffit d'ajouter la clause $a \vee d$ ou la règle $\neg d \rightarrow a$ pour que la base devienne totalement achevée.

Par défaut, le terme **achèvement** sans autre qualificatif se réfère à un achèvement total.

3.2 Achèvement partiel

Lorsqu'on dispose d'information sur les bases de faits que l'on utilisera par la suite, il est possible d'éviter l'ajout de règles qui ne seront jamais déclenchées par les faits considérés. Par ailleurs, on peut éventuellement ne souhaiter obtenir une information complète que pour certains littéraux. On parle alors d'achèvement partiel car la complétude ne sera garantie que pour certaines bases de faits et certains littéraux.

Définition 18:

On appelle **achèvement partiel** pour un ensemble de bases de faits $\mathcal{F}$ et un ensemble de littéraux $\mathcal{L}$ l'opération qui à une base $\mathcal{B}$ de formules fait correspondre une base (de règles ou de clauses selon le type de chaînage avant utilisé) $Achvt(\mathcal{B})$ telle que

- $\mathcal{B} \equiv Achvt(\mathcal{B})$ (équivalence sémantique)
- $\forall F \in \mathcal{F},\ \forall l \in \mathcal{L},\ \mathcal{B} \cup F \models l \Leftrightarrow l \in Fwch(\mathcal{B} \cup F)$
- $\forall F \in \mathcal{F},\ \mathcal{B} \cup F \models \square \Leftrightarrow \exists l,\ \{l, \neg l\} \subseteq Fwch(\mathcal{B} \cup F)$

Exemple 19:

Soit $\mathcal{B}$ la base ci-dessous

$$\begin{cases} a \vee b \\ a \vee c \\ \neg b \vee \neg c \vee d \\ e \vee f \\ e \vee g \\ \neg f \vee \neg g \vee h \end{cases}$$

On s'intéresse uniquement aux déductions qui portent sur a et h et aux bases de faits construites avec les littéraux d et e. On a donc $\mathcal{L} = \{a, \neg a, h, \neg h\}$ et

$$\mathcal{F} = \{\varnothing, \{d\}, \{\neg d\}, \{e\}, \{\neg e\}, \{d, e\}, \{\neg d, e\}, \{d, \neg e\}, \{\neg d, \neg e\}\}$$

$\mathcal{B}$ n'est pas partiellement achevée pour $\mathcal{F}$ et $\mathcal{L}$ puisque $a \notin Fwch(\mathcal{B} \cup \{\neg d\})$ alors que $\mathcal{B} \cup \{\neg d\} \models a$.

Il suffit d'ajouter la clause $a \vee d$ ou la règle $\neg d \rightarrow a$ pour que la base devienne partiellement achevée pour $\mathcal{F}$ et $\mathcal{L}$.

Par rapport à un achèvement total, on fait ici l'économie de l'ajout de $e \vee h$ puisque cette clause n'est pas nécessaire au chaînage avant pour effectuer les déductions qui nous intéressent.

3.3 Forces et faiblesses

Les avantages que présente l'achèvement sont nombreux

- La complétude des inférences effectuées en chaînage avant est garantie sur la base achevée.
- La complexité des interrogations futures devient linéaire par rapport à la taille de la base achevée, puisque le chaînage avant est de complexité linéaire[1].
- L'exécution du chaînage avant sur la base achevée est en général plus rapide que l'utilisation d'une méthode complète de production (cf. le chapitre 8).
- Il s'agit d'une compilation de la base. En effet, la base achevée peut être utilisée pour toute base de faits. Le temps investi dans la compilation peut donc être amorti après un certain nombre de requêtes.
- Il s'agit également d'une compilation logique, ce qui signifie que le résultat de l'achèvement est purement logique. Il peut donc être retravaillé d'un point de vue opérationnel (factorisation des prémisses des règles, compilation directe du chaînage avant en assembleur, ...) ou librement réutilisé par d'autres algorithmes de logique si besoin est (algorithmes d'interrogations comme l'unit-réfutation par exemple).
- On conserve le chaînage avant comme moteur d'inférence. L'achèvement est donc compatible avec les systèmes experts existants.
- Il n'y a aucune restriction sur la syntaxe des formules acceptées par l'achèvement. En effet, la compilation les transforme en clauses ou règles que le chaînage avant sait alors utiliser. Il devient possible de décrire très librement la connaissance en utilisant par exemple des règles à conclusion dubitative $a \rightarrow b \vee c$ ou des formules plus complexes telles que $AtLeast(3, \{a, b, c, d\})$ (au moins trois des littéraux a, b, c, d sont vrais).
- L'utilisateur peut retrouver sa connaissance et celle qui a été ajoutée sous la forme de règles qu'il sait facilement comprendre. L'achèvement permet également de simplifier la connaissance en éliminant des règles redondantes.

La faiblesse principale de l'achèvement est sa complexité en temps et en espace. Cependant, toute méthode de compilation conservant l'équivalence est confrontée à ce type de problème. Dans le pire des cas, une base achevée est exponentiellement plus grande que la base initiale (la complexité en temps est donc également exponentielle dans le pire des cas). Cependant, cette explosion se produit surtout pour des bases qui codent des problèmes mathématiques ardus. Or, l'achèvement est essentiellement destiné aux bases codant des problèmes réels décrits par des experts. De telles bases n'ont pas la complexité des problèmes mathématiques, et le problème d'explosion de la base achevé est ainsi peu fréquent sur des problèmes réels.

[1] sur un modèle de machine à mémoire à accès aléatoire où le coût d'accès à une case mémoire ne dépend pas de l'adresse de cette case.

Chapitre 4

Les autres compilations logiques

Nous présentons assez succinctement les compilations logiques qui se rapprochent le plus de l'achèvement. En particulier, nous passons sous silence des méthodes plus éloignées telles que [CS92] et [MT93] (entre autres). On notera cependant que chacune de ces compilations est utilisée par leur auteur pour la complétude de méthodes d'interrogation et non de production. Une méthode d'interrogation calcule une fonction de la forme

$$\mathcal{B}, F, l \longmapsto \mathbb{V} \text{ ou } \mathbb{F}$$

tandis qu'une méthode de production calcule une fonction de la forme

$$\mathcal{B}, F \longmapsto L \subseteq Lit(\mathcal{B})$$

Ces méthodes de compilation peuvent être classifiées en deux grandes catégories. La première est celle des compilations qui préservent l'équivalence, c'est à dire celles qui construisent une base sémantiquement équivalente à la base initiale. La seconde regroupe les compilations qui ne produisent pas une base équivalente mais seulement une base qui permet d'effectuer certaines des inférences que permettait la base initiale. On parle alors de compilation approchée, puisque la base compilée approxime la base initiale.

4.1 Les compilations préservant l'équivalence

4.1.1 La vivification

Entre autres sujets, [Dal95] étudie l'algorithme BCP (qui n'est autre que le chaînage avant sur des clauses). En particulier, il étend cet algorithme pour ne plus se restreindre à des bases en CNF. L'algorithme résultant est appelé FP (Fact Propagation) et permet de produire des faits impliqués par une formule quelconque et un ensemble de faits. On note $vdash_{FP}$ les inférences réalisées par cet algorithme. Cet algorithme est plus puissant que BCP (il infère plus de faits sur une formule f que BCP sur la forme clausale de f) mais demeure incomplet. Sa complexité est quadratique, tandis que BCP est de complexité linéaire [Dal92a][Dal95][Gén96].

[Dal95] présente alors une méthode de compilation logique nommée ***vivification*** [Lev86] qu'il construit par une méthode de point fixe (de son propre aveu très inefficace). Cet méthode est basée sur l'opérateur $T_{\mathcal{B},k}(S)$ défini ainsi

$$T_{\mathcal{B},k}(S) = \{C | (C \text{ clause}) \wedge (length(C) \leq k) \wedge (\mathcal{B} \cup S \cup \neg C \vdash_{FP} \square)\}$$

On note alors $lfp(T_{\mathcal{B},k})$ le plus petit point fixe de cet opérateur. Soit $Viv(\mathcal{B}, k) = \mathcal{B} \cup lfp(T_{\mathcal{B},k})$. Il existe un plus petit k appelé ***intricacy*** tel que $\forall C$, $(\mathcal{B} \models C) \Leftrightarrow (Viv(\mathcal{B}, k) \cup \neg C \vdash_{FP} \square)$. La base $Viv(\mathcal{B}, intricacy(\mathcal{B}))$ est alors une base pour laquelle l'algorithme FP est complet. Elle constitue donc une compilation de la base $\mathcal{B}$ pour l'algorithme d'inférence FP.

4.1.2 FPI_0, FPI_1 et FPI_2

[dV94] présente trois méthodes de compilation dédiées à l'unit-réfutation. La première, FPI_0 (*Filtered Prime Implicate*), consiste schématiquement à calculer l'ensemble des impliqués premiers par une méthode de résolution et à ne conserver que ceux qui sont obtenus par résolution avec fusion (filtrage). On notera que cette méthode correspond au théorème 30 de [Mat91]. FPI_1 est similaire à FPI_0. Elle calcule l'ensemble des impliqués premiers et ne retient que ceux qui sont obtenus par résolution avec fusion et dont un littéral fusionné sert ultérieurement de pivot. Enfin, FPI_2 est un raffinement de FPI_0 où l'on interdit d'effectuer des résolutions entre deux clauses de Horn.

L'unit-refutation est une méthode d'inférence qui paraît au premier abord plus puissante que le chaînage avant. Par exemple, elle permet de prouver a à partir de $\{a \vee b, a \vee \neg b\}$ tandis que le chaînage avant ne le peut pas. Cependant, les deux méthodes sont toutes les deux basées sur l'unit-propagation et leur pouvoir d'inférence intrinsèque est en fait le même. Seul l'usage qui est fait de l'unit-propagation les différencie. Dans le cas de l'unit-refutation, on ajoute la négation du fait à prouver à la base avant de rechercher par unit-propagation si cela aboutit à une contradiction. En chaînage avant en revanche, on ne connaît pas à l'avance le fait à prouver et l'unit-propagation ne peut donc pas faire usage de la négation du but à prouver. L'unit-réfutation apparaît donc plus puissante uniquement parce qu'elle dispose d'une information supplémentaire qui est le but à prouver et qu'elle utilise à la manière du chaînage arrière.

FPI_0 produit des bases qui sont complètes aussi bien pour le chaînage avant que pour l'unit-réfutation (puisque toute base complète pour le chaînage avant l'est aussi pour l'unit-refutation). La méthode d'achèvement par cycle est très proche de FPI_0 puisque toutes deux sont basées sur la recherche des résolutions avec fusions. La différence essentielle est que la méthode des cycles recherche activement les résolvantes avec fusion tandis que FPI_0 effectue un filtrage passif. De plus, la méthode des cycles incorpore à la recherche des clauses à ajouter le mécanisme de simplification par chaînage avant, alors que FPI_0 ne contient aucun mécanisme de ce type. Par opposition, FPI_2 contient un tel mécanisme puisque les résolutions entre clauses de Horn y sont interdites. Ce mécanisme est à la fois moins et plus puissant que la simplification par chaînage avant. D'un coté, la simplification par chaînage avant s'applique aussi bien à des clauses Horn que non Horn et est en ce sens plus générale. De l'autre, la simplification par chaînage avant ne permet pas d'éliminer toutes les résolvantes de clauses de Horn. En effet, certaines sont indispensables à la complétude du chaînage avant (ex : $\{a \rightarrow b,\ a \rightarrow c,\ b \wedge c \rightarrow d\}$). De ce fait, FPI_2 ne produit pas une base achevée. Les bases produites par FPI_1 ne le sont pas davantage.

4.1.3 Compilation via les impliquants

La méthode de compilation proposée par [Sch96a] consiste essentiellement à calculer une couverture d'impliquants premiers $\mathcal{I}$ de la base $\mathcal{B}$ en utilisant l'algorithme de Davis et

Putnam et le critère de minimisation de [CC96b]. L'algorithme d'interrogation utilisé est le suivant.

1: **function** Interrogation(C : clause) : booléen
2: /* retourne $\mathbb{V}$ si $\mathcal{B} \models C$ et $\mathbb{F}$ sinon */
3: **for all** impliquant I de $\mathcal{I}$ **do**
4: **if** $I \cap C = \varnothing$ **then**
5: **return** $\mathbb{F}$ /* $\mathcal{B} \not\models C$ */
6: **end if**
7: **end for**
8: **return** $\mathbb{V}$ /* $\mathcal{B} \models C$ */
9: **end function**

Cet algorithme revient en fait à vérifier que C peut être produit par traversée de matrice [Mat91]. Tout l'intérêt de la méthode provient de ce que ce calcul est effectué lors de la requête et est de complexité polynomiale.

Bien que [Sch96a] ne le propose pas, il est facile de transformer cet algorithme en un algorithme de production. En effet, l'intersection des impliquants qui ne contredisent pas les faits forme l'ensemble des faits impliqués.

1: **function** Production(F : ensemble de faits) : ensemble de faits
2: /* retourne l'ensemble des littéraux impliqués par $\mathcal{B} \cup F$ */
3: **return** $\bigcap_{I \in \mathcal{I},\ I \cap \neg F = \varnothing} I$
4: **end function**

Bien que très séduisante pour des bases qui ont peu de modèles, cette approche se révèle très vite inutilisable sur des bases telles qu'un additionneur binaire qui ont un nombre exponentiel de modèles.

4.1.4 Impliqués et impliquants premiers modulo une théorie

Les compilations proposées par [Mar95], [MS96] se fondent sur la notion de localité et d'impliqué modulo une théorie (**theory prime implicate**). Une base $\mathcal{B}$ est dite locale pour l'interrogation si tout impliqué de la base est en fait un impliqué d'une clause de la base prise isolément. Un **impliqué modulo une théorie** Φ d'une base $\mathcal{B}$ est une clause C telle que $\mathcal{B} \cup \Phi \models C$ ce que l'on note $\mathcal{B} \models_\Phi C$. Une clause C_1 **subsume modulo une théorie** Φ une formule C_2 si $C_1 \models_\Phi C_2$. La subsumption est stricte si $C_2 \not\models_\Phi C_1$. Un **impliqué premier modulo une théorie** d'une base est un impliqué modulo la théorie qui n'est strictement subsumé modulo la théorie par aucun autre. La notion d'impliqué premier modulo une théorie est donc une généralisation de la notion habituelle d'impliqué premier. On note $\boldsymbol{TPI(\mathcal{B}, \Phi)}$ l'ensemble des impliqués premiers de $\mathcal{B}$ modulo la théorie Φ. Une méthode possible [Mar95] pour calculer l'ensemble des impliqués premiers modulo une théorie est d'effectuer une saturation par résolution en utilisant comme critère de subsomption la subsomption modulo la théorie. [MS96] présente un autre algorithme basé sur l'usage d'arbres de décision binaires [Bry86]. L'intérêt de cette dernière méthode est de permettre la suppression au plus tôt des impliqués qui ne sont pas premiers modulo la théorie.

L'ensemble des impliqués premiers modulo la théorie Φ forme une base locale pour $\models_\Phi$. La compilation d'une base $\mathcal{B}$ consiste à choisir une théorie Φ telle que $\mathcal{B} \models \Phi$ et telle

que, quelles que soient C_1 et C_2, on sait dire en temps polynomial si $C_1 \models_\Phi C_2$ ou non. Une fois cette théorie choisie, la compilation revient à calculer $TPI(\mathcal{B}, \Phi)$. L'algorithme d'interrogation à utiliser est le suivant :

```
1: function Interrogation(C : clause) : booléen
2:   /* retourne V si B |= C et F sinon */
3:   for all clause C' de TPI(B, Φ) do
4:     if C' |=_Φ C then
5:       return V /* B |= C */
6:     end if
7:   end for
8:   return F /* B |≠ C */
9: end function
```

Le choix de la théorie Φ a bien sûr une influence considérable sur la taille de $TPI(\mathcal{B}, \Phi)$. [Mar95] propose d'énumérer les théories Φ possibles et de retenir celle qui produit le plus petit ensemble $TPI(\mathcal{B}, \Phi)$ (énumération guidée par diverses heuristiques et effectuée en temps borné).

[MM96] étend les travaux de [Sch96a] en présentant une compilation basée sur le calcul d'une couverture d'impliquants modulo une théorie. Un **impliquant modulo une théorie** Φ d'une base $\mathcal{B}$ est une conjonction de littéraux M telle que $M \models_\Phi \mathcal{B}$. Un **impliquant premier modulo une théorie** d'une base est un impliquant modulo la théorie qui n'est strictement subsumé modulo la théorie par aucun autre. La notion d'impliquant premier modulo une théorie est donc une généralisation de la notion habituelle d'impliquant premier. Une **couverture d'impliquants modulo une théorie** Φ d'une base $\mathcal{B}$ est un ensemble d'impliquants modulo Φ de $\mathcal{B}$ qui lui est équivalent modulo Φ. On note $\boldsymbol{TIC(\mathcal{B}, \Phi)}$ une couverture d'impliquants de $\mathcal{B}$ modulo la théorie Φ. La méthode de compilation proposée par [MM96] commence par identifier Ψ et Φ tels que $\mathcal{B} \equiv \Psi \wedge \Phi$ et tels que, quels que soient M et C, on sache dire en temps polynomial si $M \models_\Phi C$ ou non. La compilation consiste alors à calculer une couverture d'impliquants de Ψ modulo la théorie Φ. L'interrogation est fondée sur le fait que $(\mathcal{B} \models C) \Leftrightarrow (\forall M \in TIC(\Psi, \Phi),\ M \models_\Phi C)$. L'algorithme qui en découle est le suivant.

```
1: function Interrogation(C : clause) : booléen
2:   /* retourne V si B |= C et F sinon */
3:   for all impliquant I de TIC(Ψ, Φ) do
4:     if I |≠_Φ C then
5:       return F /* B |≠ C */
6:     end if
7:   end for
8:   return V /* B |= C */
9: end function
```

L'algorithme proposé par [MM96] pour calculer la couverture d'impliquants modulo la théorie est une variante de l'algorithme de [Sch96a] utilisant $\models_\Phi$ au lieu de $\models$.

4.2 Les compilations ne préservant pas l'équivalence

La méthode de compilation proposée dans [SK91],[SK94] consiste à approximer une base $\mathcal{B}$ par deux bases de clauses de Horn $\mathcal{B}_{lb}, \mathcal{B}_{ub}$ telles que $\mathcal{B}_{lb} \models \mathcal{B} \models \mathcal{B}_{ub}$. Ces deux bases sont considérées comme des minorants ($\mathcal{B}_{lb}$, *lower bound*) et majorants ($\mathcal{B}_{ub}$, *upper bound*) qui permettent d'encadrer la base $\mathcal{B}$. Cet encadrement est utilisé dans l'algorithme d'interrogation comme suit :

```
1: function Interrogation(α : formule en CNF) : booléen
2:   /* retourne V si B ⊨ α et F sinon */
3:   if B_ub ⊨ α then
4:     return V /* B ⊨ α */
5:   end if
6:
7:   if B_lb ⊭ α then
8:     return F /* B ⊭ α */
9:   end if
10:
11:  utiliser une méthode complète pour répondre
12: end function
```

Puisque les minorants et majorants ne sont qu'une approximation de la base, il est nécessaire de recourir à une méthode complète d'interrogation dans certains cas. Cependant, puisque le test de satisfiabilité d'un ensemble de clauses de Horn est de complexité linéaire[2], l'usage de ces bornes se révèle payant même lorsque l'on choisit des bornes très larges comme par exemple celles constituées uniquement de clauses unitaires. Il est intéressant de noter que, bien qu'il s'agisse d'une compilation approchée, la complexité du calcul du meilleur encadrement ne peut être polynomiale si P≠NP [SK94].

Il est naturel de rechercher le meilleur encadrement possible, c'est à dire de chercher deux bases $\mathcal{B}_{glb}$ (*greatest lower bound*) et $\mathcal{B}_{lub}$ (*lowest upper bound*) telles que $\mathcal{B}_{glb} \models \mathcal{B} \models \mathcal{B}_{lub}$ et $\nexists \mathcal{B}'$ base de clauses de Horn, $((\mathcal{B}_{glb} \models \mathcal{B}' \models \mathcal{B}) \wedge (\mathcal{B}' \not\models \mathcal{B}_{glb})) \vee ((\mathcal{B} \models \mathcal{B}' \models \mathcal{B}_{lub}) \wedge (\mathcal{B}' \not\models \mathcal{B}_{lub}))$. Très schématiquement, la base $\mathcal{B}_{glb}$ est obtenue en retirant certains littéraux des clauses non Horn de $\mathcal{B}$ de manière à ce qu'elles devienne des clauses de Horn. La base $\mathcal{B}_{lub}$ est quant à elle constituée par un sous-ensemble des clauses de Horn qui sont des impliqués premiers de $\mathcal{B}$ (ou un ensemble qui lui est équivalent). La taille de $\mathcal{B}_{lub}$ peut être exponentielle en fonction de la taille de $\mathcal{B}$. [KS92] permet de remédier dans certains cas à ce problème en augmentant le vocabulaire de la base. Cependant, à moins que NP⊆P/poly [SK94], il n'est pas toujours possible d'obtenir une borne $\mathcal{B}_{lub}$ de taille polynomiale. L'algorithme utilisé dans [SK91] pour effectuer le calcul de $\mathcal{B}_{lub}$ est en fait une saturation par résolution avec la condition que chaque étape de résolution fasse intervenir au moins une clause non Horn. [dV96] raffine ce dernier algorithme en imposant comme condition que chaque étape de résolution fasse intervenir exactement une clause non Horn, ce qui permet d'obtenir dans certains cas une borne supérieure exponentiellement plus petite qu'avec l'algorithme de [SK94].

Cette technique d'encadrement de la base peut s'étendre au premier ordre. Le calcul de $\mathcal{B}_{glb}$ reste essentiellement le même. En revanche, celui de $\mathcal{B}_{lub}$ pose problème car cette borne n'est pas nécessairement une base finie. L'algorithme de [dV96] se révèle alors plus intéressant car il termine plus souvent que celui de [SK94].

[2] donc de complexité nettement inférieure à celle d'une méthode complète

Deuxième partie

L'achèvement en calcul propositionnel

Chapitre 5

Les approches sémantiques

On peut se convaincre très rapidement que l'ensemble des clauses à ajouter à une base pour obtenir une base achevée équivalente à la base initiale est un sous-ensemble des impliqués premiers de la base. En effet, comme l'on impose que la base achevée soit sémantiquement équivalente à la base initiale, on ne peut ajouter que des impliqués de cette dernière. D'autre part, ajouter tous les impliqués permet d'obtenir un achèvement puisque l'on ajoute alors une description de toutes les déductions possibles. On remarque ensuite qu'ajouter un impliqué subsumé est inutile car la clause qui le subsume sera d'abord utilisée par le chaînage avant et rendra caduque la clause subsumée. Enfin, on voit immédiatement sur l'exemple $a \rightarrow b, b \rightarrow c$ qu'ajouter l'impliqué premier $a \rightarrow c$ ne modifie en rien la complétude du chaînage avant. Les preuves de ces affirmations peuvent être trouvées dans [Mat91] ou [MD94b].

Une méthode sémantique classique de calcul d'impliqués premiers d'une base de clauses est de calculer la distributivité du $\vee$ sur le $\wedge$ sur l'ensemble des modèles de la base. Le moyen le plus efficace de calculer l'ensemble des modèles est d'utiliser un arbre sémantique de Davis et Putnam. Le calcul de la distributivité quant à lui peut s'effectuer par ce que nous appelons un arbre sémantique clausal.

5.1 L'arbre sémantique de Davis et Putnam

L'arbre sémantique de Davis et Putnam est une application de la stratégie diviser pour régner. L'idée consiste à choisir un littéral l de la base et à rechercher les modèles qui contiennent l puis ceux qui contiennent $\neg l$. Rechercher les modèles qui contiennent l revient à faire l'hypothèse que l est interprété à $\mathbb{V}$. On peut donc éliminer de la base les clauses qui contiennent ce littéral car elles représentent des contraintes satisfaites. On peut par ailleurs retirer les occurrences de $\neg l$ des clauses de la base puisque $\neg l$ ne pourra plus satisfaire ces clauses. La fonction Simplifier ci-dessous effectue cette opération.

1: **function** Simplifier($\mathcal{B}$:base, l :littéral) :base
2: /* renvoie la simplification de la base $\mathcal{B}$ sous l'hypothèse l est interprété à $\mathbb{V}$ */
3: $\mathcal{B}' = \mathcal{B}$
4: retirer de $\mathcal{B}'$ toute clause contenant l
5: retirer de chaque clause de $\mathcal{B}'$ toute occurrence de $\neg l$
6: **return** $\mathcal{B}'$
7: **end function**

On obtient alors une base simplifiée sur laquelle on peut poursuivre la recherche de modèle. Cette recherche s'arrête quand on aboutit à l'un des deux cas triviaux :

- la base est vide : elle est donc satisfaite
- la base contient $\square$: elle est donc insatisfiable

La procédure récursive DP ci-dessous code une version de l'algorithme de Davis et Putnam [DP60][DLL62] adaptée au calcul d'un ensemble de modèles. Elle doit être appelée par DP($\mathcal{B}, \varnothing$) et permet alors d'énumérer une couverture de modèles partiels de la base.

```
1: procedure DP(B :base,M :ensemble de littéraux)
2:   if □ ∈ B then
3:     return
4:   end if
5:
6:   if B = ∅ then
7:     M est un modèle
8:     return
9:   end if
10:
11:  choisir un littéral l de B
12:
13:  DP(Simplifier(B, l),M ∪ l)
14:  DP(Simplifier(B, ¬l),M ∪ ¬l)
15: end procedure
```

Le choix du littéral de la base permettant la division du problème en deux doit être fait avec soin. Une bonne heuristique est de choisir dans les clauses les plus courtes le littéral le plus utilisé. On peut également envisager des heuristiques plus complexes telle que celle de [DABC94]. On notera également que si le littéral l choisi apparaît dans une clause unitaire, il est inutile de rechercher des modèles contenant $\neg l$ puisqu'on obtiendrait immédiatement la clause vide. Cette simple remarque permet d'éviter des étapes inutiles de backtracking. En revanche, puisque l'on recherche tous les modèles, il n'est pas possible de ne rechercher que les modèles contenant l lorsque l est pur[3]. Une autre optimisation est possible quand la base ne contient plus qu'une clause. En effet, dans ce cas, il suffit de prolonger la branche en cours de construction par chacun des éléments de la clause, ce qui permet d'aboutir à des modèles.

Un exemple d'arbre construit par cette procédure est représenté ci-dessous. Les feuilles sont étiquetées par *failure* lorsque l'on a obtenu une clause vide, par *success* si l'on a obtenu une base vide et par *success (opt)* pour les succès générés par une base ne contenant plus qu'une clause. Les arcs sont étiquetés par le littéral qui a été supposé vrai pour effectuer la simplification.

[3] Un littéral pur est un littéral dont la négation ne figure pas dans la base.

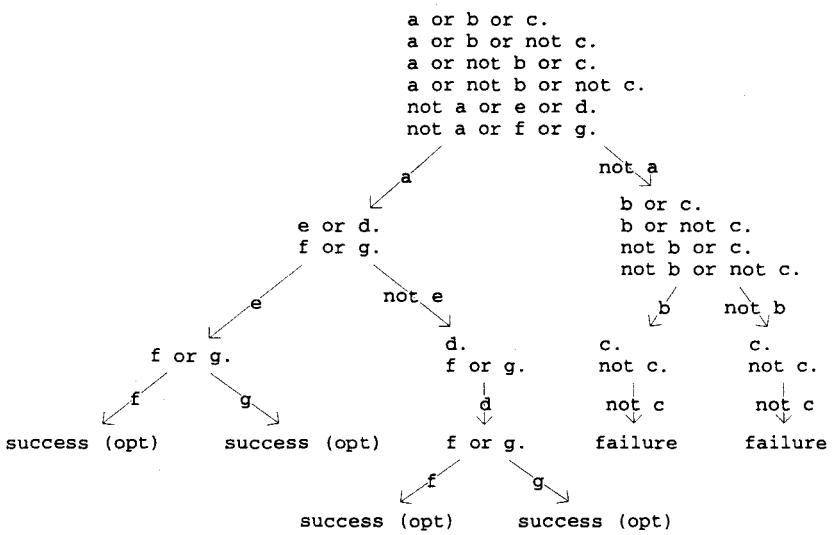


Figure 1: Exemple d'arbre de Davis et Putnam

Les modèles produits par cette procédure ne peuvent se subsumer l'un l'autre. Cependant, il ne sont pas pas nécessairement des impliquants premiers de la base.

5.2 L'arbre sémantique clausal

Par souci de clarté, nous présentons la méthode de l'arbre sémantique clausal pour un calcul d'impliquants. Son utilisation pour un calcul d'impliqués se déduit par dualité.

Si la méthode de Davis et Putnam permet de calculer efficacement un ensemble de modèles, elle s'avère inadaptée au calcul d'un ensemble d'impliquants premiers. Nous présentons donc un autre type d'arbre sémantique qui permet de calculer cet ensemble.

L'idée de l'algorithme est que pour satisfaire une base, il faut satisfaire chacune de ses clauses. Le point central de l'algorithme est alors de choisir une clause et d'énumérer les manières de la satisfaire. Il s'agit d'une démarche duale à celle de l'algorithme de Davis et Putnam où le point central était de choisir un littéral et d'énumérer les valeurs de vérité qu'il pouvait avoir. Pour énumérer les manières de satisfaire une clause, il suffit de l'ordonner et d'énumérer les littéraux qui sont les premiers à la satisfaire, c'est à dire les littéraux de la clause tels qu'il n'existe pas de littéral qui soit vrai dans le modèle et antérieur dans la clause. Cette notion de premier littéral à satisfaire une clause permet d'éviter d'énumérer plusieurs fois le même modèle. Par exemple, la clause $l_1 \vee l_2 \vee \ldots \vee l_n (l_i < l_j \Leftrightarrow i < j)$ peut être satisfaite en premier par l_1 ou l_2 ou ... ou l_n. Cela revient à dire que (l_1 est vrai) ou (l_2 est vrai et l_1 ne l'est pas) ou (l_3 est vrai et ni l_1, ni l_2 ne le sont) ou ... ou (l_n est vrai et ni l_1, ni l_2,..., ni l_{n-1} ne le sont). Il est important de noter que préciser qu'un littéral n'est pas vrai dans le modèle ne signifie pas qu'il y soit faux. En effet, on s'intéresse à des modèles partiels et un littéral peut figurer dans le modèle sous forme positive ou négative, mais aussi ne pas y figurer du tout. Le moyen le plus simple de préciser qu'un littéral ne doit plus être vrai dans le modèle est de le retirer de la base. La fonction Supprimer ci-dessous effectue cette opération.

1: **function** Supprimer($\mathcal{B}$:base, L :ensemble de littéraux) :base
2: $\mathcal{B}' = \mathcal{B}$
3: retirer de chaque clause de $\mathcal{B}'$ toute occurrence des littéraux de L
4: **return** $\mathcal{B}'$
5: **end function**

On définit alors la procédure récursive CST (Clausal Semantic Tree). Elle doit être appelée par CST($\mathcal{B}, \varnothing$) et permet d'énumérer les impliquants (premiers ou non) de la base.

1: **procedure** CST($\mathcal{B}$:base,M :ensemble de littéraux)
2: **if** $\square \in \mathcal{B}$ **then**
3: **return**
4: **end if**
5:
6: **if** $\mathcal{B} = \varnothing$ **then**
7: M est un impliquant
8: **return**
9: **end if**
10:
11: choisir une clause C de $\mathcal{B}$
12:
13: **for all** $l_i \in C$ **do**
14: CST(Simplifier(Supprimer($\mathcal{B}, \{l_j | l_j \in C, j < i\}), l_i$),$M \cup l$)
15: **end for**
16: **end procedure**

Comme pour l'algorithme de Davis et Putnam, le choix de la clause doit être effectué avec soin. Une bonne heuristique est de choisir parmi les clauses les plus courtes celle qui contient les littéraux les plus utilisés. Des heuristiques plus complexes peuvent être trouvées dans [CC96a]. Il est souhaitable également d'ordonner les littéraux de la clause dans l'ordre décroissant de fréquence.

Telle quelle, cette procédure calcule l'ensemble des impliquants au lieu de l'ensemble des impliquant premiers qui nous intéressent. Pour remédier à cela, il suffit d'utiliser le critère de minimalité de [CC96b]

Proposition 20: [CC96b]

Un impliquant est premier si et seulement si pour chacun des littéraux qui le composent, il existe une clause de la base qui n'est satisfaite que par ce littéral.

En effet, s'il existe un littéral l de l'impliquant tel que toute clause satisfaite par l l'est aussi par un autre littéral, alors, il est possible de supprimer l de l'impliquant pour en obtenir un autre qui le subsume. Réciproquement, si pour chacun des littéraux qui composent l'impliquant, il existe une clause de la base qui n'est satisfaite que par ce littéral, alors, il n'est pas possible d'obtenir un autre impliquant par retrait de certains littéraux. Ce critère de minimalité est d'une efficacité remarquable car il suffit de gérer des compteurs pour effectuer le test.

La procédure que nous avons décrite est très semblable à celle proposée par [CC96b]. Elle en diffère uniquement par la mise en avant de la dualité avec l'algorithme de Davis et Putnam : la procédure de Davis et Putnam sélectionne un littéral et énumère les valeurs de vérité possibles de ce littéral tandis que notre procédure sélectionne une clause puis énumère les manières de la satisfaire. En fait, excepté le critère de minimisation et diverses optimisations mineures, le principe de cette procédure est le même que celui des procédures basées sur les SE-tree [Rym94], sur le produit unioniste [CC96b] ou sur la traversée de matrice [Mat91] (elle-même basée sur [Bib81]).

Un exemple d'arbre construit par cette procédure est représenté ci-dessous. Les feuilles sont étiquetées par *failure* lorsque l'on a obtenu une clause vide, par *success* si l'on a obtenu une base vide et par *success (opt)* pour les succès générés par une base ne contenant plus qu'une clause. Les arcs sont étiquetés par le littéral qui a été supposé vrai pour effectuer la simplification.

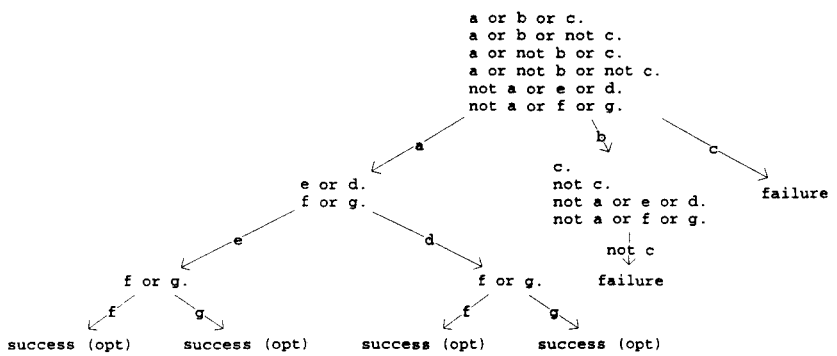


Figure 2: Exemple d'arbre sémantique clausal

5.3 La simplification par chaînage avant

Une fois calculé l'ensemble des impliqués premiers de la base, il est possible d'éliminer des impliqués qui ne sont pas nécessaires à la complétude du chaînage avant. Cette opération est cependant optionnelle, puisque la base est d'ores et déjà achevée.

Un algorithme simple pour effectuer cette simplification est celui-ci [Mat91].

for all $C \in Achvt(\mathcal{B})$ **do**
  inutile$\leftarrow \mathbb{V}$
  for all $l_1 \wedge \ldots \wedge l_n \rightarrow l$ variante de C **do**
    if $l \notin Fwch((Achvt(\mathcal{B}) - C) \cup \{l_1, \ldots, l_n\})$ **then**
      inutile$\leftarrow \mathbb{F}$
      break
    end if
  end for
  if inutile **then**
    retirer la clause C de $Achvt(\mathcal{B})$

end if

end for

Cet algorithme permet d'obtenir une base irredondante.

Définition 21:

Une base achevée est **irredondante** si et seulement si aucune clause ne peut être retirée de cette base sans perdre la propriété d'achèvement.

Chapitre 6

L'achèvement par parties

Comme le nombre d'impliqués premiers à produire est exponentiel dans le pire des cas, toute méthode de calcul d'impliqué premier a une complexité spatiale et temporelle exponentielle. Cependant, les performances de l'achèvement peuvent être grandement améliorées si l'on réussit à segmenter la base. En effet, si T est la taille de la base et n le nombre de parties qui la composent, on a en général $n.e^{T/n} \ll e^T$ comme l'illustre la courbe de la figure 3.

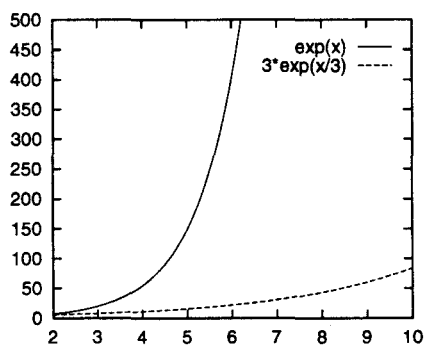


Figure 3: Comparaison d'exponentielles.

On retrouve cette stratégie de segmentation en satisfaction de contraintes [Dec89] ainsi que depuis peu pour le test de satisfiabilité [PVG96]. Il s'agit en fait d'effectuer les inférences le plus localement possible.

6.1 Définitions et théorèmes

L'achèvement par parties vise donc à réduire la complexité de la méthode classique en découpant la base par un algorithme de complexité faible pour ensuite achever les sous-bases obtenues et enfin regrouper les achevés par un algorithme de complexité également faible.

Définition 22:

On appelle **achèvement par parties** la méthode d'achèvement qui consiste à scinder une base en sous-bases qui sont alors achevées indépendamment puis regroupées.

Définition 23:

Une **sous-base** $\mathcal{B}'$ d'une base $\mathcal{B}$ sur un vocabulaire donné V est une base telle que

- $\mathcal{B}' \subseteq \mathcal{B}$
- $\forall C \in \mathcal{B}, Voc(C) \subseteq V \Leftrightarrow C \in \mathcal{B}'$

Exemple 24:

Soit la base $\{a \rightarrow b, c \rightarrow b\}$. La sous-base pour le vocabulaire $\{a, b\}$ est $\{a \rightarrow b\}$. Celle pour $\{a, b, c\}$ est la base complète, tandis que la sous-base pour le vocabulaire $\{b\}$ est la base vide.

Par souci de simplification et lorsque $\mathcal{B}'$ est non vide, on peut omettre de préciser pour quel vocabulaire $\mathcal{B}'$ est une sous-base. Dans ce cas, on considère que $V = Voc(\mathcal{B}')$.

On connaît à ce jour trois théorèmes qui permettent de segmenter une base de manière intéressante. Dans chacun de ces théorèmes, chaque $\mathcal{B}_i$ est une sous-base de $\mathcal{B}$.

Théorème fondamental de l'achèvement par parties

Soit $\mathcal{B}$ une base telle que

- $\mathcal{B} = \bigcup_i \mathcal{B}_i$
- $\forall i, j,\ i \neq j \Rightarrow Voc(\mathcal{B}_i) \cap Voc(\mathcal{B}_j) = \varnothing$.

Dans ce cas, $\bigcup_i Achvt(\mathcal{B}_i)$ est un achèvement de $\mathcal{B}$.

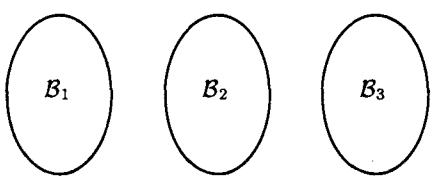


Figure 4: Illustration du théorème fondamental.

Théorème d'achèvement par parties à une clause de liaison

Soit $\mathcal{B}$ une base telle que

- $\mathcal{B} = \{C\} \cup \bigcup_i \mathcal{B}_i$
- $\forall i, j\ i \neq j \Rightarrow Voc(\mathcal{B}_i) \cap Voc(\mathcal{B}_j) = \varnothing$
- $\forall i,\ Voc(C) \cap Voc(\mathcal{B}_i) \neq \varnothing$.

Sous ces conditions, $\bigcup_i Achvt(\mathcal{B}_i \cup C)$ est un achèvement de $\mathcal{B}$.

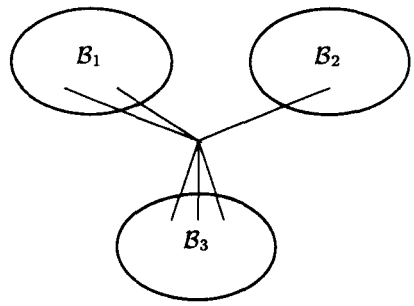


Figure 5: Illustration du théorème à une clause de liaison.

Théorème d'achèvement par parties à un atome de liaison

Soit $\mathcal{B}$ une base telle que

- $\mathcal{B} = \bigcup_i \mathcal{B}_i$
- $\forall i, j,\ \mathcal{B}_i \cap \mathcal{B}_j = \varnothing$
- $\exists a, \forall i, j,\ Voc(\mathcal{B}_i) \cap Voc(\mathcal{B}_j) = \{a\}$

Sous ces conditions, $\bigcup_i Achvt(\mathcal{B}_i)$ est un achèvement de $\mathcal{B}$.

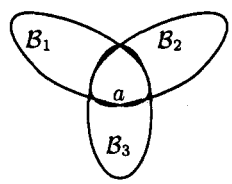


Figure 6: Illustration du théorème à un atome de liaison.

Les deux premiers théorèmes découlent des théorèmes d'achèvement total par parties de [DM93] et [MD94a]. Ces trois théorèmes sont une conséquence directe de la proposition 87 qui sera présentée ultérieurement. D'autres théorèmes d'achèvement par parties en découlent d'ailleurs, mais leur intérêt semble toutefois limité.

Il est bien sûr possible d'appliquer récursivement l'achèvement par parties aux sous-bases résultant de l'application d'un théorème. Comme chaque segmentation permet de gagner un temps parfois considérable, il convient de toujours chercher à découper les sous-bases obtenues. On recherche donc quels sont les sous-ensembles maximaux de la base qui ne peuvent être achevés par parties avec les théorèmes dont on dispose. Ces sous-ensembles forment des paquets qui devront être achevés de manière classique.

Définition 25:

On considère l'ordre entre bases défini par l'inclusion de leur vocabulaire. Précisément, une base $\mathcal{B}$ est plus petite qu'une base $\mathcal{B}'$ si $Voc(\mathcal{B}) \subset Voc(\mathcal{B}')$ ou si $Voc(\mathcal{B}) = Voc(\mathcal{B}')$ et $\mathcal{B} \subseteq \mathcal{B}'$. On appelle alors **paquet** d'une base $\mathcal{B}$, une sous-base maximale pour cet ordre et qui ne peut être décomposée par les théorèmes d'achèvement par parties.

Il convient cependant de prendre garde au théorème à un atome de liaison. En effet, celui-ci permet de séparer les clauses unitaires du reste de la base. Or, cette séparation empêche

l'utilisation de ces clauses pour simplifier la base. Il convient donc avant tout usage du théorème à un atome de liaison de propager l'information apportée par les clauses unitaires en effectuant un simple chaînage avant.

6.2 Identification des paquets

L'identification des paquets repose sur le fait que deux paquets sont reliés par au plus un atome ou une clause. Cela suggère une méthode d'identification basée sur la recherche de composantes 2-connexes dans un graphe. Les notions de théorie des graphes nécessaires à la définition de composante 2-connexe sont maintenant présentées.

Définition 26:

Un **graphe non orienté**[4] est une paire (N, A) où

- N, l'ensemble des nœuds du graphe, est un ensemble d'objets

$$\{n_1, n_2, \ldots, n_m\}$$

- A, l'ensemble des arêtes du graphe, est un ensemble de couples non ordonnés d'objets de N

$$A = \bigcup_k \{(n_{i_{2k}}, n_{i_{2k+1}})\}$$

Une **chaîne** de longueur l est une séquence de l arêtes de la forme :

$$\left((n_{i_1}, n_{i_2}), (n_{i_2}, n_{i_3}), (n_{i_3}, n_{i_4}), \ldots, (n_{i_l}, n_{i_{l+1}})\right)$$

n_{i_1} est l'extrémité initiale de la chaîne, $n_{i_{l+1}}$ son extrémité terminale.

Cette chaîne est décrite de manière équivalente[5] par la séquence des nœuds rencontrés :

$$\{n_{i_1}, n_{i_2}, n_{i_3}, n_{i_4}, \ldots, n_{i_l}, n_{i_{l+1}}\}$$

Un graphe est dit **connexe** s'il existe une chaîne entre chacun de ses nœuds.

Un **sous-graphe** (N', A') d'un graphe (N, A) est un graphe tel que

- $N' \subseteq N$
- $A' \subseteq A$
- $\forall i, j\ (i, j) \in A' \Rightarrow (i \in N') \wedge (j \in N')$

Une **composante connexe** d'un graphe G est un sous-graphe connexe de G et maximal pour l'ordre généré par la relation d'inclusion de l'ensemble des arêtes.

Un **point d'articulation** est un nœud d'un graphe G dont la suppression augmente le nombre de composantes connexes de G.

Une **composante 2-connexe** est une composante connexe d'un graphe qui ne contient pas de point d'articulation.

[4] La définition que nous donnons ici est en fait celle d'un 1-graphe non orienté. Comme tous les graphes que nous manipulerons sont des 1-graphes, nous les appellerons simplement graphe.

[5] dans le cas d'un 1-graphe en tout cas.

On construit alors un graphe dont les composantes 2-connexes sont les paquets que l'on recherche. Il est construit sur la notion d'atome et de clause comme suit :

Définition 27:

Le **graphe d'atomes** associé à une base $\mathcal{B}$ est le graphe tel que

- L'ensemble des nœuds de G est l'ensemble des atomes de $\mathcal{B}$ (**nœud atome**) *union*
  l'ensemble des identificateurs de clauses de $\mathcal{B}$ (**nœud clause**).
- L'ensemble des arêtes de G est l'ensemble des arêtes (c, a) telles que c soit l'identificateur d'une clause de $\mathcal{B}$ et a un atome figurant dans cette clause.

Exemple 28:

La base

$$\begin{cases} ① \neg a \vee b \\ ② \neg a \vee c \\ ③ \neg b \vee \neg c \vee d \\ ④ \neg d \end{cases}$$

est représentée par le graphe

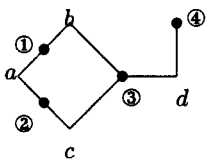


Figure 7: Exemple de graphe d'atomes.

On a choisi ici d'identifier les clauses par leur numéro et de marquer les nœuds clauses par des points. Cependant, chaque fois que cela ne cause aucune ambiguïté, on omet de préciser sur le graphe les identificateurs de clauses.

Par construction, les sous-bases qui ne peuvent être séparées par le théorème fondamental d'achèvement par parties sont des composantes connexes de ce graphe. Par ailleurs, les clauses ou atomes de liaison correspondent à des points d'articulation. Les paquets de la base sont donc bien représentés par les composantes 2-connexes du graphe.

Rechercher les composantes 2-connexes est une opération facile. L'algorithme de Tarjan [HT73] effectue cette opération en O(nombre d'arêtes). La recherche d'une segmentation a donc une complexité nettement inférieure à celle de l'achèvement classique et permet de donner tout son sens à l'achèvement par parties. Le regroupement des sous-bases achevées est lui aussi très facile puisqu'il s'agit d'une simple union.

L'algorithme d'achèvement par parties est donc le suivant.

1: /* utiliser les clauses unitaires avant application du théorème à un atome de liaison */
2: propager l'influence des clauses unitaires par chaînage avant.
3:
4: /* déterminer les paquets */
5: utiliser l'algorithme de Tarjan pour identifier les composantes 2-connexes.

6: **for all** composante 2-connexe du graphe **do**
7: créer un paquet contenant les clauses dont l'identificateur appartient à la composante
8: **end for**
9: /* achever par parties la base découpée */
10: **for all** paquet P **do**
11: achever P de manière classique
12: **end for**

Exemple 29:

Le fonctionnement de l'algorithme est le suivant. Soit la base

$$\begin{cases} a \rightarrow b \\ a \rightarrow c \\ \neg b \vee \neg c \vee d \vee e \\ d \rightarrow f \\ e \rightarrow f \\ f \rightarrow g \\ f \rightarrow h \\ g \wedge h \rightarrow i \\ j \rightarrow k \\ j \rightarrow l \\ k \wedge l \rightarrow m \end{cases}$$

Les composantes 2-connexes de son graphe d'atomes sont encerclés sur le graphe ci-dessous. Les points d'articulation sont les nœuds à la frontière de deux composantes.

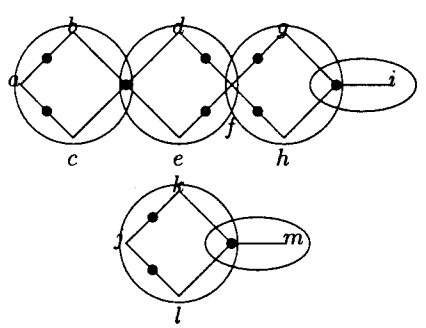


Figure 8: Exemple d'achèvement par parties.

Les paquets créés par l'algorithme sont donc

① $\begin{cases} a \rightarrow b \\ a \rightarrow c \\ \neg b \vee \neg c \vee d \vee e \end{cases}$

② $\begin{cases} \neg b \vee \neg c \vee d \vee e \\ d \rightarrow f \\ e \rightarrow f \end{cases}$

③ $\begin{cases} f \rightarrow g \\ f \rightarrow h \\ g \wedge h \rightarrow i \end{cases}$

④ $\{\ g \wedge h \rightarrow i$

⑤ $\begin{cases} j \rightarrow k \\ j \rightarrow l \\ k \wedge l \rightarrow m \end{cases}$

⑥ $\{\ k \wedge l \rightarrow m$

Chaque paquet est achevé indépendamment, ce qui provoque l'ajout à la base des clauses $\neg a \vee d \vee e$ pour le premier paquet, $\neg b \vee \neg c \vee f$ pour le deuxième, $\neg f \vee i$ pour le troisième et $\neg j \vee m$ pour le cinquième.

6.3 Résultats expérimentaux

Quelques uns des résultats expérimentaux obtenus pour l'achèvement par parties sont maintenant présentés sur un jeu de bases de tests décrit en annexe. On s'intéresse en particulier au temps nécessaire à l'achèvement par parties *(part)* comparé au temps nécessaire à un achèvement classique (i.e. par calcul d'un arbre de Davis et Putnam suivi du calcul d'une arbre sémantique clausal, sans segmentation) *(clas)*. D'autres indicateurs sont le nombre d'atomes de la base *(N. At)*, le nombre de clauses *(N. Cl)* le nombre de paquets détectés *(Pqts)* ainsi que la taille de la base qui est définie comme suit.

Définition 30:

La **taille** d'une base $\mathcal{B}$ est la somme des longueurs des clauses qu'elle contient.

Les temps sont mesurés sur une machine SUN SPARCstation 5/85 et exprimés en secondes. Ils ont été arbitrairement limités à 5 minutes, et tout dépassement est signalé par un tiret.

Toutes les bases ne permettent pas l'application des trois théorèmes d'achèvement par parties. En particulier, les bases qui codent des problèmes mathématiques complexes ne se segmentent quasiment pas. C'est également le cas de certaines bases structurées. On peut cependant constater sur le tableau ci-dessous que l'on perd en réalité très peu de temps à rechercher une segmentation de la base, même pour des bases de plus de 1000 atomes et clauses.

Base initiale					Base achevée					
Base					N. Clauses		Taille		Tps Achvt	
Nom	N. At	N. Cl	Taille	Pqts	part	clas	part	clas	part	clas
députés	16	20	62	1	109	109	405	405	0.07	0.05
logiciens	11	15	54	1	100	100	397	397	0.02	0.02
pannes	16	30	79	3	164	185	565	642	0.03	0.03
pigeon-4-5	20	74	160	1	1414	1414	8220	8220	23.62	24.85
ramsey-4	18	36	90	1	1464	1464	10278	10278	45.43	47.38
type2-1000	1002	1001	3001	2	1002	1002	3003	3003	2.73	1.77
type4-9	29	28	73	2	549	549	7507	7507	8.48	8.50

En revanche, lorsque les théorèmes peuvent être appliqués, les gains en performance sont spectaculaires. C'est en particulier le cas pour la majorité des bases structurées ainsi que pour l'additionneur binaire. Pour ce type de base, l'achèvement passe d'une complexité exponentielle en temps à une complexité polynomiale (aussi bien en temps qu'en espace) grâce aux théorèmes d'achèvement par parties. Ces résultats s'expliquent par le fait que les segmentations effectuées permettent d'éviter le calcul inutile de la transitivité de certaines règles, ce qui se retrouve sur la taille des bases achevées.

Le cas de l'additionneur binaire est très représentatif de ce phénomène. Par construction, un additionneur binaire de n bits a 4^n modèles. Les méthodes sémantiques de calcul d'impliqués premiers deviennent donc très rapidement inutilisables. Les méthodes syntaxiques ne font d'ailleurs guère mieux. En revanche, lorsque l'on utilise l'achèvement par parties, il est possible de découper cet additionneur de n bits en n additionneurs de 1 bit. En effet, chaque cellule ne partage qu'un atome avec sa voisine et le théorème à un atome de liaison s'applique donc.

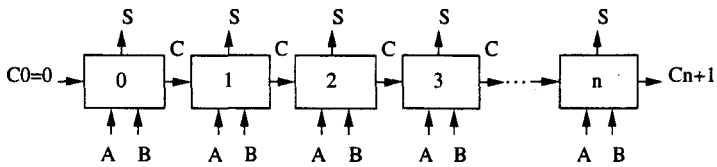


Figure 9: Un additionneur binaire de $n + 1$ bits

L'achèvement par parties permet dans ce cas de passer d'un achèvement exponentiel en temps à un achèvement linéaire. En effet, la recherche des composantes 2-connexes du graphe est linéaire. On trouve n composantes connexes (chaque additionneur de un bit). L'achèvement de l'ensemble de ces composantes est donc linéaire par rapport à la taille de l'additionneur (et donc de la base). Par opposition, les autres méthodes d'achèvement (sémantiques ou syntaxiques) qui n'effectuent pas ces segmentations seront nécessairement exponentielles en temps.

Il devient alors possible d'achever des bases de plus de 1600 atomes et 5600 clauses en moins d'une minute (adder-400).

Base initiale					Base achevée					
Base					N. Clauses		Taille		Tps Achvt	
Nom	N. At	N. Cl	Taille	Pqts	part	clas	part	clas	part	clas
adder-3	13	43	150	3	36	303	107	1358	0.00	0.64
adder-4	17	57	200	4	50	1037	151	5509	0.01	32.20
adder-400	1601	5601	20000	400	5594	–	17575	–	36.04	–
type1-150	150	149	298	150	149	11175	298	22350	0.03	137.72
type1-850	850	849	1698	850	849	–	1698	–	5.88	–
type3-13	27	14	40	27	14	8205	40	114714	0.00	174.88
type3-500	1001	501	1501	1001	501	–	1501	–	2.85	–
type5-6	25	19	49	13	25	753	61	6615	0.00	14.68
type5-7	29	22	57	15	29	2215	71	22662	0.00	151.85
type5-400	1601	1201	3201	801	1601	–	4001	–	15.40	–
type6-7	29	28	63	15	35	301	77	700	0.00	0.28
type6-12	49	48	108	25	60	876	132	2040	0.00	23.88
type6-15	61	60	135	31	75	1365	165	3180	0.00	288.28
type6-300	1201	1200	2700	601	1500	–	3300	–	9.42	–
type7-5	26	20	50	16	25	539	60	2417	0.00	16.33
type7-6	31	24	60	19	30	906	72	4494	0.00	275.60
type7-250	1251	1000	2500	751	1250	–	3000	–	7.52	–

Les graphes ci-dessous présentent une comparaison des temps d'achèvement pour la méthode classique et la méthode par parties pour les bases type1 et type6, en fonction du nombre de clauses de la base.

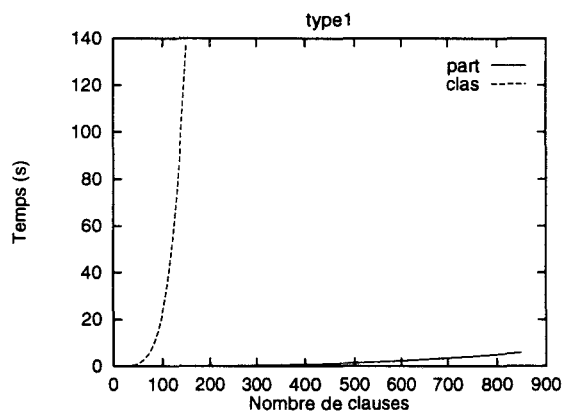


Figure 10: Performances de l'achèvement par parties sur la base type1

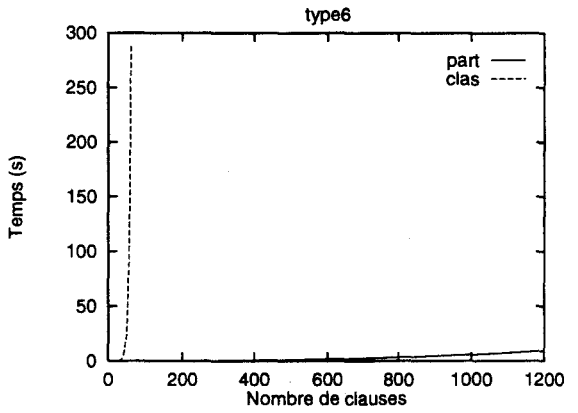


Figure 11: Performances de l'achèvement par parties sur la base type6

On constate une similarité certaine avec la comparaison des exponentielles de la figure 3. En fait, le gain apporté par l'achèvement par parties est encore plus important car le nombre de sous-bases croît avec le nombre de clauses.

L'intérêt de l'achèvement par parties est si grand qu'on souhaiterait bien évidemment obtenir d'autres théorèmes pour pouvoir effectuer toujours plus de segmentations. Cela semble malheureusement difficile. En effet, lorsque l'on découpe une base en sous-bases qui partagent plus d'un atome ou d'une clause à leur interface, il n'est en général plus possible d'obtenir un regroupement polynomial des achevés. Cela ne signifie pas que les seuls théorèmes d'achèvement par parties (avec regroupement polynomial) sont ceux que nous avons cité. On pourrait énoncer par exemple que si deux sous-bases n'ont que des littéraux purs en communs, alors il est possible de les achever séparément (corollaire de la proposition 87). Toutefois un tel théorème semble rarement pouvoir s'appliquer. Il semble en fait plus fructueux de rechercher des segmentations à un niveau plus fin, ce qui est le rôle de l'achèvement par cycles.

Chapitre 7

Les approches syntaxiques

Les méthodes syntaxiques d'achèvement sont toutes fondées sur le principe de résolution de Robinson [Rob65].

Définition 31:

Soient deux clauses C_1 et C_2 et un littéral l tels que $l \in C_1$ et $\neg l \in C_2$. La **résolution** de C_1 et C_2 selon le **pivot** l est la production de la clause $(C_1 - \{l\}) \cup (C_2 - \{\neg l\})$ appelée **résolvante** de C_1 et C_2.

Cette règle d'inférence peut être utilisée pour produire les impliqués d'une base.

7.1 La saturation par résolution

La méthode syntaxique la plus simple pour le calcul des impliqués premiers d'une base est d'effectuer une saturation par résolution [CL73].

Définition 32:

Soit $\mathcal{B}$ une base de clauses et $S(\mathcal{B}) = \mathcal{B} \cup \{\text{ensemble des résolvantes de } \mathcal{B}\}$. La **saturation par résolution** de $\mathcal{B}$ est la première base $S^n(\mathcal{B})$ telle que $S^{n+1}(\mathcal{B}) = S^n(\mathcal{B})$.

Il s'agit donc de produire toutes les résolvantes possibles. Cependant, les tautologies et clauses subsumées sont en général de peu d'intérêt. Deux résultats de base permettent de les éliminer au plus tôt. Nous rappelons ces résultats car il sont vitaux pour la méthode des cycles.

Proposition 33:

Soient A, B, C trois clauses non tautologiques.

Si A subsume B, alors la résolvante de A et C subsume la résolvante de B et C si ces deux résolvantes ne sont pas des tautologies.

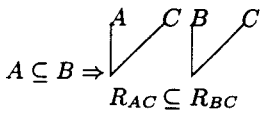


Figure 12: Schématisation

Il est donc inutile de calculer les résolvantes d'une clause subsumée, puisqu'elles sont elles-même subsumées ou tautologiques.

Démonstration 34:

$A \cap \neg C$ est un singleton qui contient p le littéral pivot de la résolution (si ce n'est pas un singleton, la résolvante est une tautologie ce qui contredit les hypothèses).

$B \cap \neg C$ est aussi un singleton (sinon la résolvante serait une tautologie). Or, puisque $A \subseteq B$ (car A subsume B), on a forcément $p \in B \cap \neg C$ et donc $B \cap \neg C = \{p\}$.

De ce fait, puisque la résolvante sur B a même pivot et que $A \subseteq B$, on en déduit que la résolvante avec A subsume la résolvante avec B (en effet, $A \subseteq B \Rightarrow (A - \{p\}) \cup (C - \{\neg p\}) \subseteq (B - \{p\}) \cup (C - \{\neg p\})$)

Proposition 35:

Soit T une tautologie et C une clause.

La résolvante de T et C est soit une clause subsumée, soit une tautologie.

Il est donc inutile de calculer de telles résolvantes.

Démonstration 36:

Soit $B = T \cap \neg T$ l'ensemble des littéraux tautologiques de T. Soit p le littéral de C utilisé comme pivot de la résolution.

Deux cas sont possibles :

- $p \in B$
  Soit $T' = T - \{p, \neg p\}$.
  La résolvante est $(C - \{p\}) \cup (T - \{\neg p\}) = C \cup T'$ qui est donc subsumée par C (p est retiré de C puis rajouté par l'union avec T).
- $p \notin B$
  Dans ce cas, la résolvante contient tous les littéraux tautologiques de B et est donc une tautologie.

On peut donc utiliser l'algorithme ci-dessous pour calculer l'ensemble des impliqués premiers par saturation par résolution.

1: $N \leftarrow \mathcal{B}$ /* N est l'ensemble des clauses ajoutées à la dernière étape de la saturation. N est toujours inclus dans $\mathcal{B}$ */
2: **repeat**
3: $N' \leftarrow \varnothing$ /* N' est l'ensemble des clauses ajoutées à l'étape courante de la saturation */
4: **for all** $C_1 \in \mathcal{B}$ **do**
5: **for all** $C_2 \in N$ tel que C_2 figure après C_1 dans $\mathcal{B}$ **do**

6: /* la condition d'ordre permet d'éviter de traiter deux fois un même couple de clause */
7: **if** C_1 et C_2 ont une résolvante non tautologique R **then**
8: **if** R subsumée par une clause de $\mathcal{B} \cup N \cup N'$ **then**
9: continue
10: **end if**
11: Retirer de $\mathcal{B}$, N, N' les clauses subsumées par R
12: Ajouter R à N'
13: **end if**
14: **end for**
15: **end for**
16: $\mathcal{B} \leftarrow \mathcal{B} \cup N'$
17: $N \leftarrow N'$
18: **until** $N = \varnothing$

Cette méthode est assez inefficace et on lui préfère la méthode de Tison [Tis67], méthode reprise dans les algorithmes [dV94][KT90]

1: **for all** $a \in Voc(\mathcal{B})$ **do**
2: **for all** $C_1 \in \mathcal{B}$ tel que $a \in C_1$ **do**
3: **for all** $C_2 \in \mathcal{B}$ tel que $\neg a \in C_2$ **do**
4: Soit R la résolvante sur a de C_1 et C_2
5:
6: **if** R est une tautologie **then**
7: continue
8: **end if**
9:
10: **if** R subsumée par une clause de $\mathcal{B}$ **then**
11: continue
12: **end if**
13:
14: Retirer de $\mathcal{B}$ toute clause subsumée par R
15: Ajouter R à la base $\mathcal{B}$
16: **end for**
17: **end for**
18: **end for**

Cependant, il s'agit toujours d'une méthode de calcul de tous les impliqués premiers alors que certains sont inutiles pour l'achèvement. La méthode d'achèvement par cycles remédie à ce problème.

7.2 L'achèvement par cycles

L'achèvement par cycles est une méthode syntaxique d'achèvement qui permet d'identifier finement les clauses qui manquent pour obtenir des inférences complètes. Cette méthode est en particulier plus fine que l'achèvement par parties puisque ce dernier est une conséquence des théorèmes d'achèvement par cycles. La méthode consiste à identifier, par recherche des cycles d'un graphe, les résolvantes linéaires input avec fusion, qui sont en général des clauses qu'il faut ajouter.

La démarche adoptée est la suivante. Nous rappelons d'abord la notion de résolution linéaire input et celle de résolution avec fusion. Nous montrons alors une condition nécessaire et suffisante d'achèvement basée sur ce concept. Nous exhibons alors un graphe dont certains cycles représentent des résolutions linéaires input avec fusion. Après avoir défini plusieurs catégories de cycles, nous montrons l'inutilité des cycles ambigus puis celle des cycles élémentaires. Nous présentons alors une version naïve de l'algorithme issu de ces résultats, puis montrons comment la simplification par chaînage avant peut être incorporée à cet algorithme. Nous donnons alors un exemple d'utilisation originale de la méthode pour conclure sur les points forts et faibles de la méthode.

7.2.1 Résolution linéaire input avec fusion

Les quelques notions classiques nécessaires à la compréhension de la méthode sont d'abord rappelées.

Définition 37:

Une résolution entre deux clauses C_1 et C_2 est dite avec **fusion** si $C_1 \cap C_2 \neq \varnothing$. Les littéraux de $C_1 \cap C_2$ sont dits fusionnés. On note $\underline{l}$ un littéral fusionné.

Définition 38:

Une **résolution linéaire input** L à partir d'un ensemble $\mathcal{B}$ de clauses est un couple de séquences de clauses $((C_0, C_1, \ldots C_n), (R_0, R_1, \ldots R_n))$ tel que

- $\forall i, C_i \in \mathcal{B}$
- $R_0 = C_0$
- $\forall i > 0, R_i$ est une résolvante de R_{i-1} avec C_i

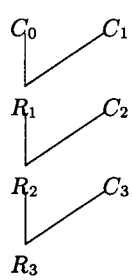


Figure 13: Exemple de résolution linéaire input

Les clauses C_i sont appelées **clauses latérales**, les R_i, **clauses centrales**. C_0 est la **racine** de la résolution linéaire input et R_n la **résolvante** de L.

Il faut noter que, contrairement à la résolution linéaire en général, on s'interdit dans une résolution linéaire input de réutiliser une clause centrale comme clause latérale. De ce fait, la résolution linéaire input n'est complète ni pour la réfutation, ni pour le calcul des impliqués premiers.

Exemple 39:

Il est impossible de dériver la clause vide par résolution linéaire input à partir de

$$\begin{cases} a \vee b \\ a \vee \neg b \\ \neg a \vee b \\ \neg a \vee \neg b \end{cases}$$

Cette incomplétude est contournée lorsqu'on effectue une saturation de la base par résolution linéaire input, c'est à dire lorsque, chaque fois qu'une résolvante est produite, on construit toutes les résolutions linéaires input qui l'utilisent, de la même manière qu'on effectue une saturation par résolution.

Définition 40:

Soit $\mathcal{B}$ une base de clauses et $S(\mathcal{B}) = \mathcal{B} \cup \{$ensemble des résolvantes finales des résolutions linéaires input de $\mathcal{B}\}$. La **saturation par résolution linéaire input** de $\mathcal{B}$ est la première base $S^n(\mathcal{B})$ telle que $S^{n+1}(\mathcal{B}) = S^n(\mathcal{B})$.

Proposition 41:

Tout impliqué premier d'une base est dans sa saturation par résolution linéaire input.

Démonstration 42:

Soit C un impliqué premier de $\mathcal{B}$. On sait que la résolution linéaire de [Lov78] est complète pour le calcul des impliqués premiers [MR72][Mat91]. Il existe donc au moins une résolution linéaire non input L de clauses centrales $(R_0, R_1, \ldots R_n)$ qui permet de produire C.

Soit $S = (R_{r_1}, R_{r_2}, \ldots, R_{r_n})$ la séquence des clauses centrales (racine exceptée) de L qui servent également de clauses latérales, séquence décrite dans l'ordre d'apparition en tant que clause centrale dans L. Chaque clause de S va permettre de découper L en tronçons, comme suit.

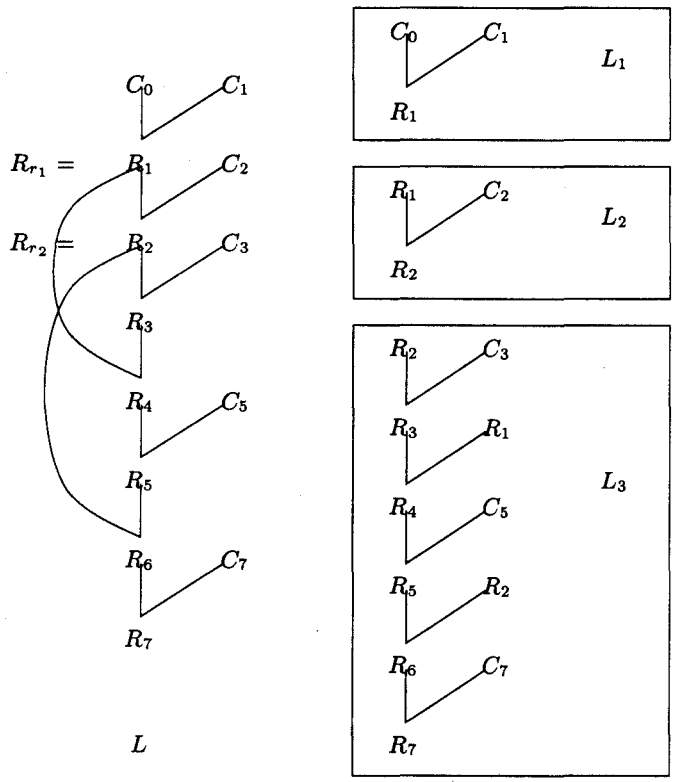


Figure 14: Segmentation d'une résolution linéaire en résolutions linéaires input

En posant $R_{r_0} = R_0$ et $R_{r_{n+1}} = R_n$, soit L_i la sous-résolution linéaire de L dont les clauses centrales sont $(R_{r_{i-1}}, R_{r_{i-1}+1}, \ldots, R_{r_i})$. Par construction, L_i ne contient pas de clause centrale qui lui serve également de clause latérale. De plus, chaque L_i est une résolution linéaire à partir de $\mathcal{B} \cup \bigcup_{j<i} R_{r_j}$. Enfin, L_{r_n+1} produit C.

Il apparaît donc que la clause C peut s'obtenir par une succession $(L_1, L_2, \ldots, L_{r_n+1})$ de résolutions linéaires input telles que

- chaque L_i produit R_{r_i}
- chaque L_i est une résolution linéaire input à partir de $\mathcal{B} \cup \bigcup_{j<i} R_{r_j}$

On en déduit que C est dans la saturation par résolution linéaire input de $\mathcal{B}$.

7.2.2 Condition nécessaire et suffisante d'achèvement

On sait déjà [MD90b] que l'ensemble des impliqués premiers d'une base forme un achèvement de celle-ci et que seuls certains d'entre eux sont nécessaires. On caractérise ci-dessous les impliqués indispensables à l'achèvement en se fondant sur la notion de résolution avec fusion.

Un littéral fusionné est en général un littéral que le chaînage avant ne sait pas produire. Par exemple, il ne sait pas inférer b de $\{\neg a, p \vee b \vee a, \neg p \vee b\}$ parce que b est fusionné dans l'impliqué $a \vee b$. Cependant, un littéral fusionné dans un impliqué peut malgré tout être produit par chaînage avant, à condition toutefois que ce soit en utilisant un autre impliqué qui lui ne comportera pas de fusion sur ce littéral.

Exemple 43:

La base

$$\begin{cases} p \vee a \vee b \\ \neg p \vee a \vee c \\ q \vee b \vee a \\ \neg q \vee b \vee c \end{cases}$$

est achevée. En effet, le seul impliqué premier qu'il faudrait ajouter est $a \vee b \vee c$. Cependant, bien que les deux résolutions qui le produisent contiennent des fusions, il est inutile d'ajouter cette clause, parce que le littéral fusionné n'est jamais le même.

La proposition ci-dessous généralise cet exemple.

Proposition 44: Condition nécessaire et suffisante d'achèvement

Une base $\mathcal{B}$ est totalement achevée si et seulement si $\forall C$ impliqué premier de $\mathcal{B}$, $\forall l \in C$, C peut être obtenue par au moins une résolution linéaire input L à partir des clauses de $\mathcal{B}$ telle que L ne contient pas de fusion sur l, ni de résolution avec pour pivot un littéral fusionné.

Démonstration 45:

On peut d'abord remarquer que, bien que très similaire, cette proposition n'est pas une conséquence du théorème 30 de [Mat91]. En effet, ce dernier ne permet pas d'éliminer $a \vee b \vee c$ dans l'exemple précédent tandis que ce théorème l'autorise.

- ($\Leftarrow$)
  On suppose que $\forall C$ impliqué premier de $\mathcal{B}$, $\forall l \in C$, C peut être obtenue par au moins une résolution linéaire input L à partir des clauses de $\mathcal{B}$ telle que L ne contient pas de fusion sur l, ni de résolution avec pour pivot un littéral fusionné.
  Soient F une base de faits et l un littéral tels que $\mathcal{B} \cup F \models l$. On sait que l'ensemble des impliqués premiers de la base est un achevé. Il existe donc une séquence d'impliqués premiers $I_1, \ldots, I_n$ tels que $I_1 \cup F \models l_1$, $I_2 \cup F \cup \{l_1\} \models l_2, \ldots, I_n \cup F \cup \{l_1, l_2, \ldots, l_{n-1}\} \models l$. Il suffit de montrer que sous les conditions du théorème, chacune de ces déductions peut être effectuée par chaînage avant.
  Soient (I, F', l) l'un des triplets (impliqué, base de faits, littéral déduit) figurant dans cette chaîne de déductions.
  Par hypothèse, I peut être obtenu par au moins une résolution linéaire input à partir de $\mathcal{B}$ sans fusion sur l ou sur les pivots des résolutions. Puisque $I \cup F' \models l$, on a nécessairement $\neg(I - \{l\}) \subseteq F'$.
  Soient C_c et C_l les clauses respectivement centrale et latérale, parentes de I dans L. On note p le pivot de cette dernière résolution tel que $p \in C_c$.
  Puisqu'il n'y a pas fusion sur l à la dernière étape de résolution, les deux cas ci-dessous sont exclusifs et exhaustifs :
  - $l \in C_c$
    Par définition de la résolution, tous les littéraux de C_l à l'exception de $\neg p$ figurent dans I. Comme $l \notin C_l$, on déduit que les littéraux $\neg(C_l - \{\neg p\})$ sont dans la base de faits. Le chaînage avant peut donc utiliser la clause $C_l \in \mathcal{B}$ pour prouver $\neg p$. À ce moment, la négation de chaque littéral de C_c — à l'exception de l — se trouve dans la base de faits. Deux cas sont possibles

- C_c est la racine de la résolution linéaire
  Dans ce cas, $C_c \in \mathcal{B}$ et le chaînage avant peut prouver l.
- C_c n'est pas la racine de la résolution linéaire
  On recommence le raisonnement sur les parents de C_c. En utilisant les mêmes arguments, on prouve par récurrence que, puisque l n'est pas fusionné dans C_c et que les pivots des résolutions précédentes ne le sont pas non plus, l peut être produit par chaînage avant.

- $l \in C_l$
  Tous les littéraux de C_c à l'exception de p se retrouvent dans C. Comme l n'est pas dans C_c, on en déduit que la négation de chaque littéral de C_c — à l'exception de p — est dans la base de faits. Deux cas sont possibles
  - C_c est la racine de la résolution linéaire
    Dans ce cas, $C_c \in \mathcal{B}$ et le chaînage avant peut prouver p.
  - C_c n'est pas la racine de la résolution linéaire
    On recommence le raisonnement sur les parents de C_c. En utilisant les mêmes arguments, on prouve par récurrence que, puisque p n'est pas fusionné dans C_c et que les pivots des résolutions précédentes ne le sont pas non plus, p peut être produit par chaînage avant.

  Une fois que p est produit, il est clair que $\neg(C_l - \{l\})$ est inclus dans la base de fait et le chaînage avant peut déduire l.

- $(\Rightarrow)$
  Par hypothèse, la base est totalement achevée. Supposons qu'il existe un littéral l et un impliqué premier I tels que toutes les résolutions linéaires L produisant I contiennent soit une fusion de l, soit une fusion sur un des pivots. Soit m ce littéral fusionné dans L. On s'intéresse à la base de faits $F = \neg(I - \{m\})$. Clairement, $\mathcal{B} \cup F \models m$. On peut alors simplifier L en effectuant les résolutions unitaires avec les faits au plus tôt. Nécessairement, L contiendra alors une résolution entre des clauses de la forme $p \vee m$ et $\neg p \vee m$ sans quoi m ne serait pas fusionné dans L. Le chaînage avant est incapable de produire m à partir de ces deux clauses, puisqu'il ne voit que des clauses binaires. Il est donc incapable de produire l et la base n'est donc pas achevée. Contradiction.

Corollaire :

Soit C une clause que l'achèvement doit ajouter pour assurer la complétude. Si l'on souhaite obtenir un achèvement sous forme de base de règles, seule doivent être ajoutées à la base les variantes de C ayant pour conclusion un littéral l tel que $\nexists$ une résolution linéaire input L produisant C et telle que l n'est pas fusionné et aucun littéral fusionné ne sert de pivot dans cette résolution.

Donc, pour achever une base, il suffit d'ajouter les impliqués avec fusion obtenus par résolution linéaire input. Cela peut évidemment se faire par saturation par résolution linéaire input. Cependant, on peut montrer qu'il est inutile de conserver à chaque étape de la saturation les clauses sans fusion issues d'une résolution linéaire input.

Proposition 46:

Soit L une résolution linéaire input à partir d'un ensemble de clauses $\mathcal{B}_L$, utilisant une clause R provenant d'une résolution linéaire input L' à partir d'un ensemble de clauses $\mathcal{B}_{L'}$ et telle qu'il n'y a aucune fusion dans L'.

L peut alors se réécrire en une seule résolution linéaire input L'' à partir de $\mathcal{B}_L \cup \mathcal{B}_{L'}$ fournissant une résolvante qui subsume au sens large celle de L.

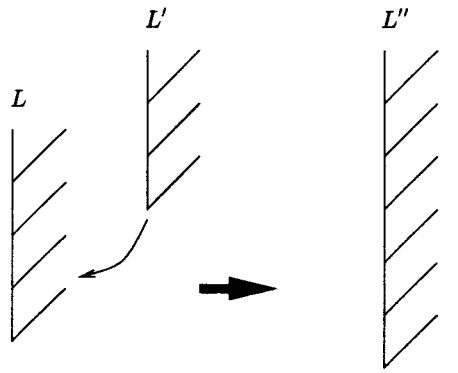


Figure 15: Réécriture de deux résolutions dont l'une est sans fusion en une seule

Démonstration 47:

On prouve que lorsque l'on effectue l'étape de réécriture ci-dessous la résolvante obtenue subsume au sens large C, et ce, à condition qu'il n'y ait pas fusion lors de la résolution entre $C^1_{L'}$ et $C^2_{L'}$.

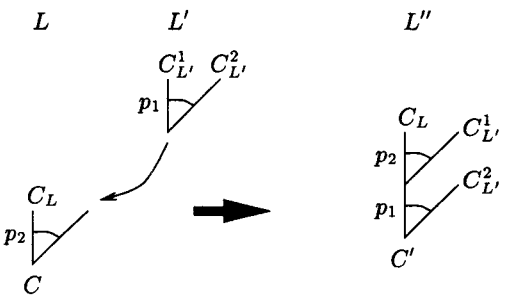


Figure 16: Étape de réécriture

En procédant alors par récurrence, on obtient le résultat annoncé, puisque chaque étape réduit la taille de L'.

Soit p_2 le littéral de C_L qui sert de pivot. On numérote les clauses de L' de sorte que $\neg p_2 \in C^1_{L'}$. Soit p_1 le littéral de $C^1_{L'}$ qui sert de pivot avec $C^2_{L'}$.

Si l'on retrace les résolutions dans la version non réécrite, on obtient que la résolvante de $C^1_{L'}$ et $C^2_{L'}$ est $(C^1_{L'} - \{p_1\}) \cup (C^2_{L'} - \{\neg p_1\})$. La résolution avec C_L donne alors $(C_L - \{p_2\}) \cup (((C^1_{L'} - \{p_1\}) \cup (C^2_{L'} - \{\neg p_1\}) - \{\neg p2\})$

Dans la version réécrite, la première résolution donne $(C_L - \{p_2\}) \cup (C^1_{L'} - \{\neg p_2\})$ et la seconde $(C^2_{L'} - \{\neg p_1\}) \cup ((C_L - \{p_2\}) \cup (C^1_{L'} - \{\neg p_2\}) - \{p_1\})$. Comme il n'y a aucune fusion dans L', et que $\neg p_2 \in C^1_{L'}$, on conclut que $\neg p_2 \notin C^2_{L'}$.

La résolvante non réécrite s'exprime donc par

$$(C_L - \{p_2\}) \cup (C^1_{L'} - \{p_1, \neg p_2\}) \cup (C^2_{L'} - \{\neg p_1\})$$

La résolvante réécrite s'écrit

$$(C_L - \{p_1, p_2\}) \cup (C^1_{L'} - \{p_1, \neg p_2\}) \cup (C^2_{L'} - \{\neg p_1\})$$

Il apparaît alors sur ces expressions que la résolvante réécrite subsume (au sens large) la résolvante non réécrite (cf le terme concernant C_L).

Note 48:

L et L'' peuvent ne pas donner les mêmes clauses, comme dans l'exemple suivant :

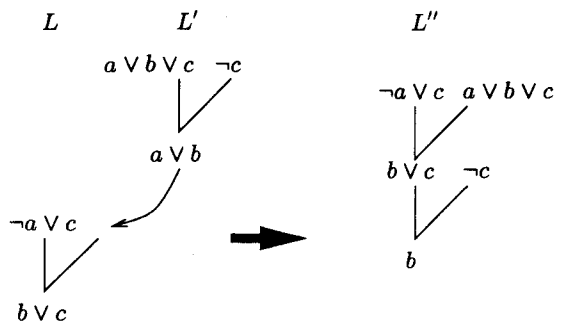


Figure 17: Exemple de réécriture donnant un résultat différent

Il n'y a par ailleurs aucune assurance que les fusions dans L'' soient les mêmes que dans L.

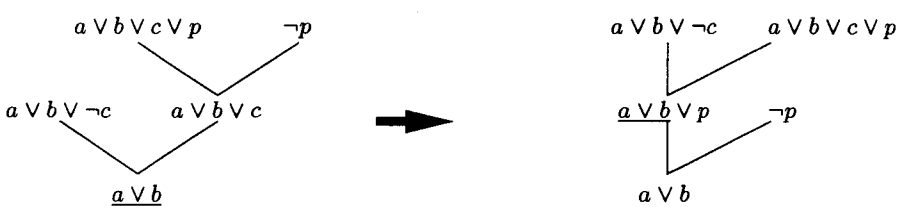


Figure 18: Exemple de réécriture modifiant les fusions

Cela n'a pas d'influence sur la complétude du chaînage avant sur la base achevée. Seules les clauses ajoutées à la base seront différentes.

7.2.3 Traduction en termes de graphes

On étudie maintenant comment les concepts de résolution linéaire input et de résolution avec fusion se traduisent en terme de graphe pour aboutir à une caractérisation élégante des clauses nécessaires à l'achèvement. Contrairement au graphe utilisé dans l'achèvement par parties, le graphe que l'on utilise prend en compte la notion de littéral.

Définition 49:

On appelle **graphe de littéraux** associé à une base $\mathcal{B}$ le graphe tel que

- L'ensemble des nœuds du graphe est constitué par
  - l'ensemble des atomes de $\mathcal{B}$ et leur négation (**nœud littéral**),
  union
  - l'ensemble des identificateurs de clauses de $\mathcal{B}$ (**nœud clause**).

- L'ensemble des arêtes du graphe est
  - l'ensemble des (C, l) tels que l est un littéral contenu dans la clause d'identificateur C, (**arête clause**)

  union
  - l'ensemble des $(a, \neg a)$ pour tout atome a (**arête pivot**).

On représente ce graphe en coloriant différemment les arêtes pivots et les arêtes clauses. On choisit ici de représenter les arêtes pivots par des pointillés (- - -) et les arêtes clauses par une ligne continue (—). Les nœuds clauses sont marqués par des points. De plus, on omet de préciser les identificateurs des nœuds clauses quand il n'y a pas d'ambiguïté.

Exemple 50:

La base

$$\begin{cases} ① \; a \vee b \\ ② \; a \vee c \\ ③ \; \neg b \vee \neg c \vee d \\ ④ \; \neg d \end{cases}$$

est représentée par le graphe

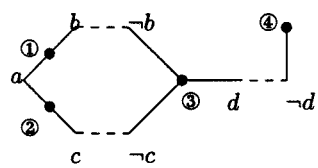


Figure 19: Exemple de graphe de littéraux.

Les arêtes pivots du graphe représentent de possibles pivots de résolution, tout comme les liens dans les graphes de connections [Kow75]. Contrairement au graphe choisi ici, les graphes de connections ne permettent pas une détection simple des fusions car les littéraux figurant dans plusieurs clauses y sont dupliqués comme on peut le voir sur cet exemple.

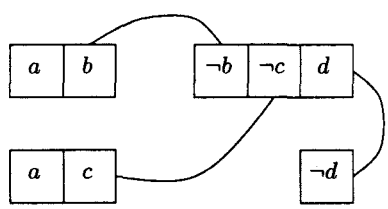


Figure 20: Exemple de graphe de connexion.

On définit d'abord sur ce graphe les chaînes qui représentent des résolutions linéaires input.

Définition 51:

Une **chaîne de résolution** est une chaîne du graphe de littéraux de la forme

$$\underbrace{l_{L1}\text{–}C_1\text{–}l_{R1}} \cdots \underbrace{l_{L2}\text{–}C_2\text{–}l_{R2}} \cdots\cdots \underbrace{l_{Ln}\text{–}C_n\text{–}l_{Rn}}$$

où chaque l_{Li}, l_{Ri} est un littéral de la base, C_i un identificateur de clause de la base tels que

$$\forall i, \begin{cases} l_{Li} \in C_i \\ l_{Ri} \in C_i \\ l_{Li} = \neg l_{R(i-1)} \\ l_{Li} \neq l_{Ri} \end{cases}$$

.

Note 52:

La condition $l_{Li} \neq l_{Ri}$ sert à interdire d'utiliser un même littéral comme pivot dans deux résolutions différentes consécutives. En effet, le littéral pivot disparaît de la résolvante et ne peut resservir de pivot.

À chaque chaîne de résolution peut être associée une résolution linéaire input. Par exemple, il suffit de lire les clauses d'une chaîne d'une extrémité à l'autre, pour construire une résolution linéaire input où les clauses apparaissent dans le même ordre et où les pivots des résolutions sont ceux indiqués par les arêtes pivots.

Mais en réalité, une chaîne de résolution retranscrit plusieurs résolutions linéaires input. On peut choisir comme clause racine n'importe quelle clause de la chaîne. On prend alors chaque fois comme clause latérale une clause contiguë dans la chaîne aux clauses déjà incorporées dans la résolution. Dans le cas le plus général, on peut choisir à chaque étape entre une clause à la droite et une à la gauche de la sous-chaîne des clauses déjà incorporées.

Exemple 53:

La figure ci-dessous présente les quatre résolutions linéaires input associées à la chaîne

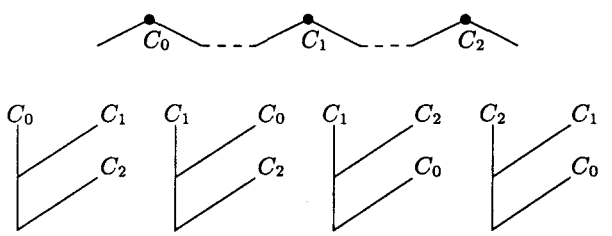


Figure 21: Diverses résolutions associées à une chaîne

Définition 54:

Une **résolution linéaire input associée à une chaîne de résolution** C est une résolution linéaire input telle que

- la clause racine de L est une clause de C
- la clause latérale C_i de L est l'une des clauses de C qui sont reliés à la sous-chaîne constituée par les clauses latérales ou centrales précédentes par une arête pivot, arête qui désigne le pivot à utiliser pour la résolution avec C_i.

Il va de soi que toute chaîne de résolution a au moins une résolution linéaire input associée. En revanche, la réciproque est fausse.

Exemple 55:

Il existe des résolutions linéaires input qui ne peuvent se traduire sous forme d'une unique chaîne de résolution.

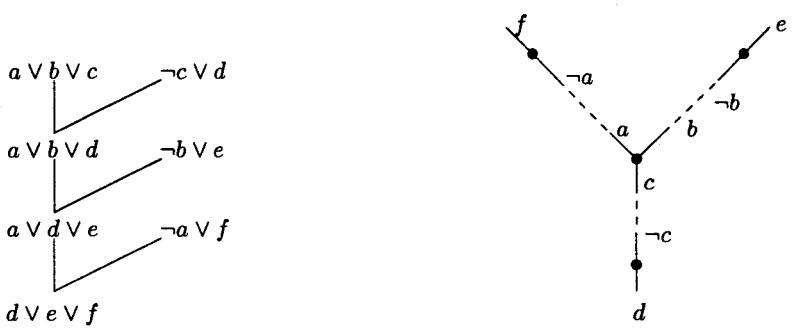


Figure 22: Exemple de résolution linéaire input qui ne correspond pas à une chaîne

Cependant, on verra que ce type de résolution linéaire ne représente pas une résolvante utile pour l'achèvement puisqu'il n'y a pas de fusion.

7.2.4 Plusieurs types de cycles

Les chaînes de résolution peuvent former des cycles.

Définition 56:

Un **cycle** est une chaîne dont l'extrémité initiale et terminale coïncident.

On distingue plusieurs types de chaînes de résolution cycliques : cycles élémentaires, ambigus, tautologiques, fusionnants ou intéressants. Les trois premiers servent uniquement dans les preuves, tandis que les deux derniers types caractérisent les clauses nécessaires à l'achèvement d'une base.

7.2.4.1 Cycle élémentaire

Le premier type de cycle est classique en théorie des graphes.

Définition 57:

Une **chaîne élémentaire** est une chaîne dans laquelle aucun nœud n'apparaît strictement plus d'une fois (sauf éventuellement à l'extrémité terminale).

Un **cycle élémentaire** est une chaîne élémentaire dont l'extrémité initiale et terminale coïncident.

On montrera par la suite qu'il est inutile de considérer ce type de cycle.

7.2.4.2 Cycle ambigu

Un cycle ou une chaîne de résolution peut représenter plusieurs résolvantes linéaires input différentes. On dit alors que la chaîne est ambiguë car on peut lui faire correspondre plusieurs résolvantes.

Définition 58:

Un cycle ou une chaîne est **ambigu** si on peut lui associer au moins deux résolvantes

différentes.

On pourra prouver que ce type de cycle n'est pas utile à l'achèvement.

7.2.4.3 Cycle tautologique

Un cycle tautologique est un cycle qui représente en général une tautologie. Il est définit comme suit.

Définition 59:

Une chaîne de résolution telle que le premier littéral de la chaîne et le dernier soient opposés est appelée **cycle tautologique**.[6]

Les cycles tautologiques ne présentent guère d'intérêt pour l'achèvement à proprement parler. Cependant, ils servent à démontrer qu'il est inutile de considérer certains types de cycles. Nous présentons maintenant quelques propriétés de ces cycles qui nous seront utiles par la suite.

Proposition 60:

Toute résolution linéaire input qui correspond à une chaîne tautologique dans laquelle aucune extrémité n'est utilisée comme pivot génère une tautologie.

Démonstration 61:

Puisqu'aucune extrémité ne sert de pivot dans une résolution, et que chacune des clauses de la chaîne figure dans la résolution linéaire, les littéraux de chaque extrémité figurent dans la résolvante qui est donc une tautologie.

Cette proposition s'applique en particulier aux chaînes élémentaires.

Corollaire

Toute résolvante linéaire input qui correspond à une chaîne tautologique élémentaire est une tautologie.

Démonstration 62:

Dans une chaîne élémentaire, les extrémités de la chaîne ne peuvent servir de pivot (car sinon la chaîne passerait deux fois sur son extrémité).

Voici un exemple de cycle tautologique qui ne produit pas une tautologie.

[6] On pourrait éventuellement définir un cycle tautologique comme une chaîne qui a pour extrémités le même nœud clause, mais cela requiert de modifier la définition de chaîne de résolution.

Exemple 63:

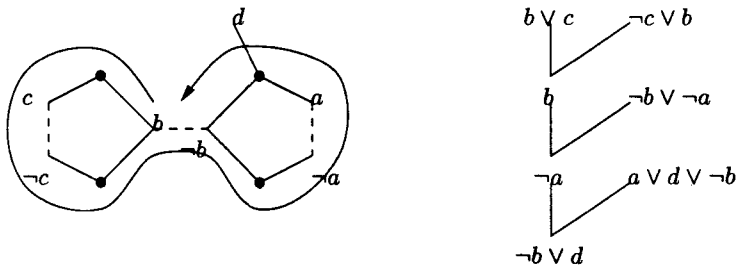


Figure 23: Exemple de résolvante non tautologique associée à un cycle tautologique

Ce cycle n'est évidemment pas élémentaire.

7.2.4.4 Cycle fusionnant

Les cycles fusionnants sont ceux qui nous intéressent directement puisqu'ils représentent en général des clauses qu'il faut ajouter pour réaliser l'achèvement d'une base. On les définit ainsi :

Définition 64:

Une chaîne de résolution telle que le premier littéral de la chaîne et le dernier soient identiques est appelée **cycle fusionnant**. L'extrémité de cette chaîne est appelée **tête de cycle**.

Le concept de cycle fusionnant est le même que celui de *tied chain* exposé dans [Esh93]. Nous préférons cependant le terme de cycle car il exprime, à notre sens, la propriété essentielle de cet objet. [Esh93] utilise les *tied chains* pour bâtir un test suffisant de complétude pour la réfutation unit. Comme le chaînage avant est une forme de résolution unit, l'usage du même concept dans les deux cas n'a rien de surprenant.

Il est clair qu'un cycle fusionnant représente une résolution linéaire input avec fusion sur la tête du cycle. La réciproque est vraie, comme le montre la proposition suivante.

Proposition 65:

À toute résolvante linéaire input qui se termine par une fusion correspond au moins un cycle fusionnant.

Démonstration 66:

On démontre ce point en transformant une quelconque résolution linéaire input L en un arbre T comme suit. Soit $(C_0, C_1, \ldots, C_n)$ la séquence des clauses latérales de L. C_0 est la clause racine que l'on transforme en l'arbre T_0 suivant :

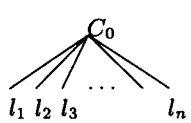


Figure 24: Première étape de la construction

L'arbre T_i s'obtient à partir de T_{i-1} en greffant la clause C_i par l'intermédiaire de son pivot à chaque feuille de T_{i-1} qui est la négation de ce pivot, comme sur l'exemple ci-dessous.

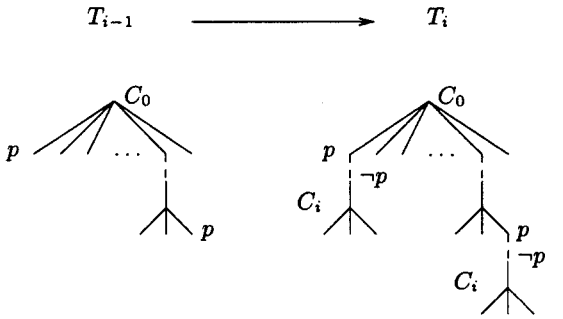


Figure 25: Ajout d'une clause à l'arbre

Les feuilles de T_i représentent les littéraux présents dans la résolvante des clauses (C_0, $C_1, \ldots, C_i$). Si L se termine par une fusion sur l, l'arbre T_n contient deux feuilles portant le littéral l. En remontant les branches portant ces feuilles jusqu'à la racine, on construit un cycle fusionnant ayant l comme extrémités.

Ce résultat est bien sûr le point essentiel de la méthode.

Note 67:

Il faut noter que si l'on parcourt le cycle fusionnant ainsi construit d'une extrémité à l'autre, on ne retrouve pas les clauses latérales dans le même ordre que dans L. Il est donc essentiel dans les démonstrations de considérer toutes les résolutions linéaires input que l'on peut associer à une chaîne, et pas seulement celles qu'on obtiendrait en lisant la chaîne d'une extrémité à l'autre.

D'autre part, toute résolution linéaire input associée à un cycle fusionnant ne se termine pas nécessairement par une fusion comme on peut le voir sur l'exemple ci-dessous. La fusion se produit bien avant d'arriver à la fin de la chaîne.

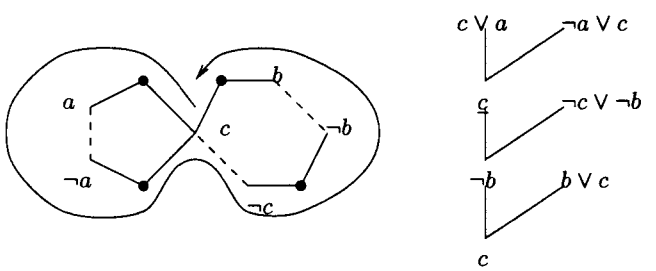


Figure 26: Exemple de cycle fusionnant dont la fusion apparaît avant la fin du cycle

Cependant, ce type de situation est construit de manière un peu artificielle. Il s'agit d'un cycle non élémentaire que nous pourrons éliminer.

7.2.4.5 Cycle intéressant

La dernière définition concerne les cycles qu'il est réellement important de rechercher. On les appelle pour cela des cycles intéressants.

Définition 68:

Un **cycle intéressant** est un cycle fusionnant, non ambigu, et élémentaire.

Le terme «intéressant» ne signifie pas que toute clause correspondant à ce cycle est utile. Il signifie seulement que ce type de cycle est susceptible de produire une clause indispensable à l'achèvement. Cependant, la clause produite pourrait être une tautologie ou une clause subsumée. Par exemple, pour la base $\{a \vee b \vee c, \neg a \vee b \vee \neg c\}$, les deux cycles intéressants du graphe génèrent une tautologie.

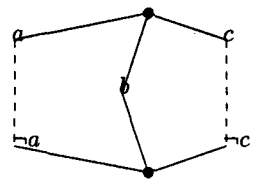


Figure 27: Exemple de cycle intéressant tautologique

On pourrait bien sûr raffiner la notion de cycle intéressant pour éliminer tautologies et clauses subsumées. Cependant, cela ne semble pas réellement utile compte tenu de la facilité avec laquelle on peut tester si une clause est une tautologie et de la non-localité du critère de subsomption.

7.2.5 Inutilité des cycles ambigus

Nous montrons maintenant qu'il est inutile de considérer les cycles ambigus. En calcul propositionnel, un cycle ou une chaîne ambigu est caractérisé sur le graphe par la présence de ponts.

Définition 69: pont

On appelle **pont** d'une chaîne de résolution une arête qui relie une clause de la chaîne (la **clause du pont**) à un littéral de la chaîne (le **littéral du pont**) qui n'appartient pas à la clause du pont et n'est pas l'extrémité de la chaîne.

On interdit que le littéral du pont soit une extrémité de la chaîne car, pour que la chaîne soit ambiguë, il faut que le littéral du pont serve de pivot dans une résolution. Selon l'ordre dans lequel apparaissent dans la résolution linéaire input la clause du pont et la résolution sur le littéral du pont, ce dernier littéral sera ou non présent dans la résolvante finale, permettant de construire ainsi différentes résolvantes à partir de la même chaîne.

Exemple 70:

La chaîne ci-dessous n'est pas ambiguë.

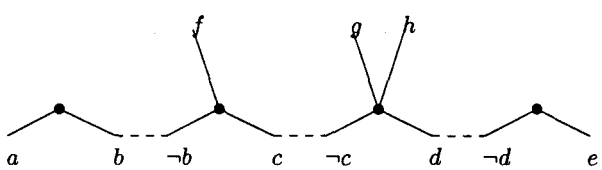


Figure 28: Exemple de chaîne non ambiguë

En revanche, celle-ci l'est du fait du pont représenté en gras.

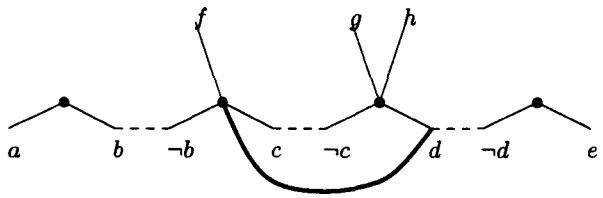


Figure 29: Exemple de chaîne ambiguë

On peut lui associer deux résolvantes linéaires input différentes :

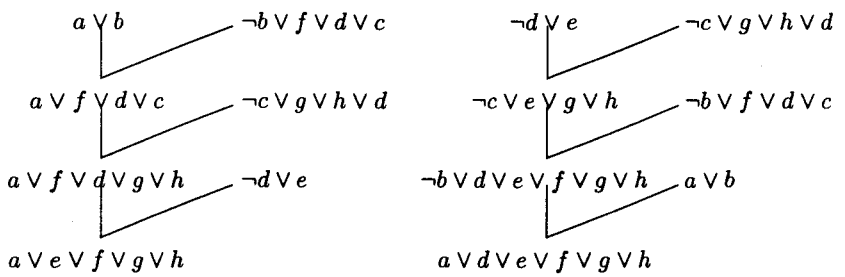


Figure 30: Deux résolvantes différentes associées à une chaîne ambiguë

Proposition 71:

Une chaîne ambiguë est une chaîne de résolution comportant au moins un pont.

On peut également formuler cette proposition comme ceci

Proposition 72:

A toute chaîne de résolution sans pont correspond une et une seule résolvante linéaire input.

Démonstration 73:

Si la chaîne de résolution n'est pas ambiguë, les atomes qui servent de pivots lors de la résolution linéaire input apparaissent exactement deux fois dans l'ensemble des clauses intervenant dans la résolution (une sous forme positive et une sous forme négative). Ils n'apparaissent donc dans aucune des résolvantes correspondant à la chaîne, et ce, quel que soit l'ordre choisi pour l'intervention des clauses dans la résolution, puisqu'ils disparaissent quand ils servent de pivot. Par ailleurs, les littéraux des clauses qui ne servent pas de pivot ne sont jamais supprimés de la clause centrale de la résolution linéaire input. Comme on utilise toutes les clauses, ils se retrouvent tous dans toutes les résolvantes. Les littéraux qui composent une résolvante obtenue en utilisant les clauses dans un ordre donné sont donc produites par l'union pour chaque clause (dans l'ordre) des littéraux de cette clause qui ne servent pas de pivot. Comme l'union est commutative, l'ordre d'utilisation des clauses dans la résolution linéaire input ne modifie pas la résolvante obtenue.

Corollaire : résolvante linéaire associée à une chaîne de résolution non ambiguë

Soit C, une chaîne de résolution non ambiguë de la forme :

$$l_{L1}\text{–}C_1\text{–}l_{R1}\cdots l_{L2}\text{–}C_2\text{–}l_{R2}\cdots l_{L3}\cdots l_{R(n-1)}\cdots l_{Ln}\text{–}C_n\text{–}l_{Rn}$$

avec $l_{Li} = \neg l_{R(i-1)}$. L'unique résolvante linéaire qui lui correspond peut s'obtenir par

$$\{l_{L1}\} \cup \{l_{Rn}\} \cup \bigcup_i (C_i - \{l_{Li}, l_{Ri}\})$$

Graphiquement, un cycle fusionnant non ambigu ressemble à un «**oursin**». La proposition précédente exprime que l'on obtient la résolvante qui lui correspond en conservant ses «**piquants**» et sa «**bouche**» (la tête du cycle).

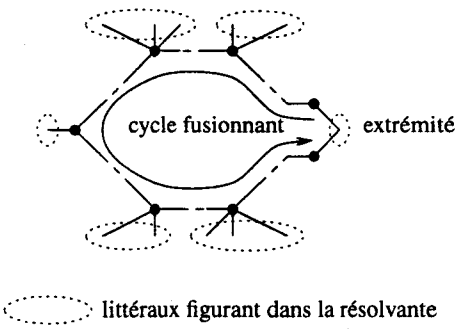


Figure 31: Un oursin

Il est nécessaire d'introduire quelques notions supplémentaires pour montrer que les cycles ambigus sont inutiles pour l'achèvement.

Définition 74:

On appelle **clauses sous le pont** les clauses d'une chaîne ambiguë situées entre les deux extrémités du pont (clause du pont comprise).

Exemple 75:

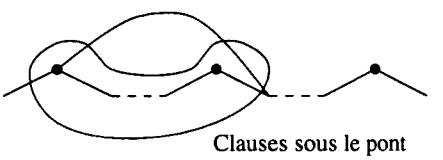


Figure 32: Exemple de clauses sous le pont

Les ponts peuvent être classés en deux catégories :

Définition 76:

On appelle **pont tautologique** un pont qui forme un cycle tautologique avec les clauses sous le pont.

On appelle **pont fusionnant** un pont qui forme un cycle fusionnant avec les clauses sous le pont.

Exemple 77:

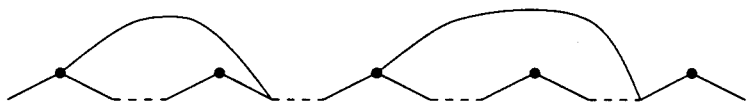


Figure 33: Exemple de pont fusionnant et de pont tautologique

Par définition, chaque pont est soit tautologique, soit fusionnant.

Définition 78:

On définit un **ordre partiel sur les ponts** comme ceci. On dit qu'un pont $\mathcal{P}_1$ est plus petit qu'un pont $\mathcal{P}_2$ si les clauses sous le pont $\mathcal{P}_1$ sont incluses dans les clauses sous le pont $\mathcal{P}_2$.

Cet ordre déterminera celui dans lequel il convient de remplacer les cycles formés par les ponts pour transformer une chaîne ambiguë en une chaîne qui ne l'est plus sans perdre de résolvante utile.

Définition 79:

Soit $\mathcal{C}$ une chaîne dont les nœuds ont été ordonnés.

On appelle **pont en avant** un pont dont la clause est supérieure au littéral.

On appelle **pont en arrière** un pont dont la clause est inférieure au littéral.

Exemple 80:

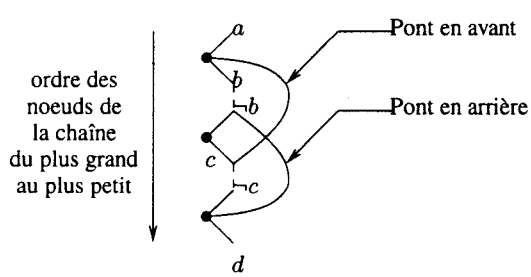


Figure 34: Exemple de pont en avant et en arrière

La résolution est commutative mais pas associative. C'est d'ailleurs la raison pour laquelle les chaînes peuvent être ambiguës.

Cependant, il est nécessaire d'établir une «certaine» forme d'associativité dans les résolutions linéaires input.

Définition 81:

Soit L une résolution linéaire input qui fait intervenir la séquence de clauses

$$(C_0, C_1, \ldots, C_{i-1}, C_i, \ldots, C_j, C_{j+1}, \ldots, C_n)$$

On appelle regroupement de C_i à C_j de L la résolution linéaire input L_r dont la séquence des clauses est

$$(C_0, C_1, \ldots, C_{i-1}, R, C_{j+1}, \ldots, C_n)$$

où R (le **regroupement**) est la résolvante de $(C_i, \ldots, C_j)$.

En général, L et L_r ne fournissent pas la même résolvante.

Proposition 82: Propriété des regroupements

Soit L une résolution linéaire input et L_r, la résolution linéaire input obtenue à partir de L en regroupant dans R des clauses consécutives $(C_i, \ldots, C_j)$.

S'il n'existe pas dans la résolvante de $(C_0, \ldots, C_{i-1})$ un littéral qui soit fusionné dans le regroupement et serve ultérieurement de pivot à l'intérieur du regroupement, alors, la résolvante de L_r subsume au sens large la résolvante de L.

Note 83:

S'il existe dans la résolvante de $(C_0, \ldots, C_{i-1})$ un littéral utilisé comme pivot dans le regroupement et qui n'est pas fusionné dans le regroupement, le regroupement ne peut être effectué. Ce cas est impossible si la résolution linéaire input correspond à une chaîne de résolution.

Démonstration 84:

Soit C la résolvante de L et C_r la résolvante de L_r. On veut montrer que $C_r \subseteq C$.

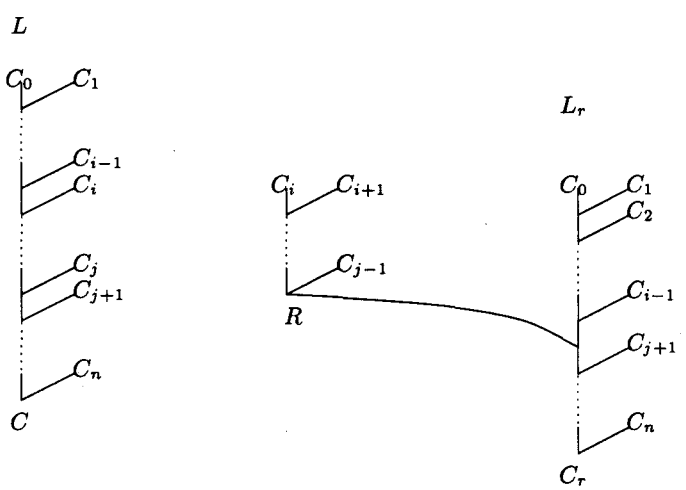


Figure 35: Exemple de regroupement

Procédons par l'absurde et supposons que $C_r \not\subseteq C$. Il existe donc un littéral l tel que $l \in C_r \wedge l \notin C$. Nécessairement, l apparaît dans une clause latérale de L_r. Soit C_l la dernière clause latérale de L_r dans laquelle figure l. Il ne peut y avoir dans L_r de résolution avec l comme pivot après l'utilisation de C_l, car sinon, l serait effacé de la résolvante.

On peut envisager deux cas :

- $C_l \neq R$
  - C_l est antérieur à R

Dans ce cas, on retrouve l dans la résolvante de $(C_0, \ldots, C_{i-1})$. Par hypothèse, l ne sert pas de pivot dans le regroupement (cf note 83). On en déduit donc que C_l n'est pas suivi d'une résolution sur l dans L. De ce fait, l figure dans C ce qui contredit l'hypothèse.

- C_l est postérieur à R

  C_l apparaît donc également dans L et n'est pas suivi d'une résolution sur l. l devrait donc figurer dans C, ce qui contredit l'hypothèse.

- $C_l = R$

  Puisque $l \in R$, on sait par hypothèse qu'il n'y a pas dans L_r après R de résolution ayant l pour pivot (sinon l serait fusionné dans le regroupement et servirait ensuite de pivot). D'autre part, si l apparaît dans R, c'est qu'il figure dans une des clauses du regroupement et qu'il ne sert pas ultérieurement de pivot dans ce regroupement. On en déduit que l figure dans une clause de L sans servir de pivot par la suite. Il devrait donc figurer dans C, ce qui contredit l'hypothèse.

Il ne peut donc exister de littéral qui soit dans C_r sans être dans C.

On montre maintenant qu'en procédant par saturation, on peut ignorer complètement les chaînes ambiguës, puisque les résolvantes qu'elles produisent seront sans intérêt ou pourront être obtenues par des chaînes non ambiguës.

Proposition 85:

Soit $\mathcal{C}$ une chaîne ambiguë et L l'une des résolutions linéaires input qui lui sont associées. La résolvante produite par L est

- soit une tautologie
- soit une clause subsumée par une autre conséquence des clauses de $\mathcal{C}$
- soit enfin une clause qui peut s'obtenir par l'utilisation d'une chaîne non ambiguë créée à partir de $\mathcal{C}$ en remplaçant les cycles créés par les ponts formant des cycles fusionnants, par les résolvantes linéaires input associées.

Démonstration 86:

Soit $(C_0, C_1, \ldots, C_n)$ la séquence selon laquelle les clauses de $\mathcal{C}$ apparaissent dans L. C_0 est donc la clause racine de L.

On considère L_i la restriction de L à ses $i+1$ premières clauses $(C_0, C_1, \ldots, C_i)$. L_i produit donc R_i, la i-ème clause centrale de L. Par convention, on considère que L_0 produit $R_0 = C_0$. R_i est donc la résolvante de R_{i-1} avec C_i (pour $i > 0$).

L_i est associée à une sous-chaîne $\mathcal{C}_i$ de $\mathcal{C}$ contenant les clauses $\{C_0, C_1, \ldots, C_i\}$

À chaque sous résolution linéaire input L_i, associée à la sous-chaîne éventuellement ambiguë $\mathcal{C}_i$, on associe une résolution linéaire L'_i qui produit la même résolvante que L_i et qui est associée à une chaîne non ambiguë $\mathcal{C}'_i$, obtenue à partir de $\mathcal{C}_i$ par transformation des cycles formés par les ponts. De plus, on impose que $\mathcal{C}'_i$ soit élémentaire.

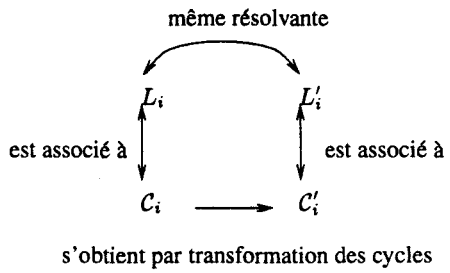


Figure 36: Relation entre les transformations

L'_i n'est pas en général une résolution linéaire input si on la construit à partir des clauses C_0 à C_n. En revanche, elle le devient si on la construit en utilisant les clauses correspondant à la transformation des cycles formés par les ponts fusionnants.

On montre par récurrence que chaque R_i est

- soit une tautologie
- soit une clause subsumée par une autre conséquence des clauses de C
- soit une clause qui peut s'obtenir par l'utilisation d'une chaîne non ambiguë élémentaire C'_i créée à partir de C_i en remplaçant les cycles formés par les ponts fusionnants par les résolvantes linéaires input associées.

Trivialement, R_0 peut s'obtenir par une chaîne non ambiguë (et élémentaire) ne contenant que la clause C_0.

Supposons maintenant que R_i satisfasse l'hypothèse de récurrence. R_i est donc

- soit une tautologie
- soit une clause subsumée par une autre conséquence des clauses de C

  Dans ces deux cas, R_{i+1} est soit une tautologie, soit une clause subsumée (cf. propositions 33 et 35).
- soit une clause qui peut s'obtenir par l'utilisation d'une chaîne non ambiguë C'_i créée à partir de C_i en remplaçant les cycles formés par les ponts fusionnants par les résolvantes linéaires input associées.

  Par hypothèse, C'_i est une chaîne non ambiguë et élémentaire. De ce fait, on peut réordonner les clauses de L'_i pour obtenir une résolution linéaire L''_i qui fournit la même résolvante que L'_i et qui correspond à une lecture des clauses de C'_i dans l'ordre, d'une extrémité de la chaîne à celle sur laquelle va porter la résolution avec C_{i+1}. Les nœuds de la chaîne sont donc ordonnés, le littéral servant de pivot avec C_{i+1} étant le plus petit.

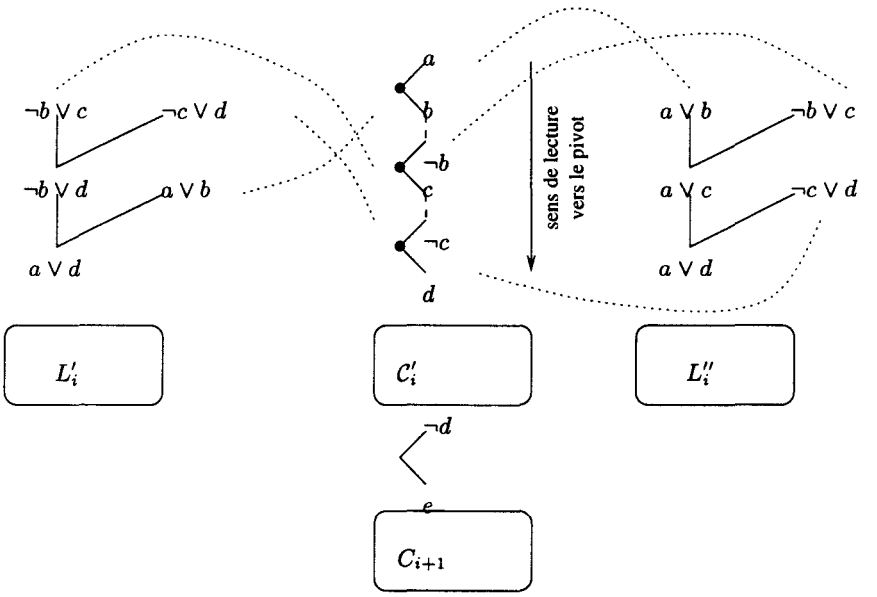


Figure 37: Exemple de réordonnancement

Ce réordonnancement permet par la suite de regrouper dans la résolution linéaire les clauses qui apparaissent sous un pont, sans être gêné par des clauses qui ne sont pas sous le pont.

On considère maintenant la chaîne $C_\cup$ formée par C'_i et C_{i+1}. On instaure un ordre sur les nœuds de cette chaîne, les nœuds supérieurs sont les plus éloignés de l'extrémité de la chaîne formée par C_{i+1}. Les nœuds de C_{i+1} sont donc les plus petits.

On envisage ci-dessous les différentes configurations de ponts. Selon les cas, la résolvante sera tautologique ou subsumée, ou bien pourra être obtenue en remplaçant les ponts fusionnants par la résolvante qu'ils représentent. On notera bien que dans ce dernier cas, la chaîne obtenue après remplacement est élémentaire si la chaîne de départ l'était.

1. $C_\cup$ ne contient pas de pont.

   l'hypothèse de récurrence reste vraie pour R_{i+1} puisque cette résolvante peut s'obtenir par la chaîne non ambiguë et élémentaire $C_\cup$ (l'absence de pont implique que $C_\cup$ soit élémentaire).

2. $C_\cup$ contient au moins un pont

   On peut d'abord remarquer que, si R_i n'est ni tautologique, ni subsumé (i.e. si l'on a su éliminer chaque pont de C'_i), les ponts de $C_\cup$ sont soit des ponts en arrière qui partent du nœud clause correspondant à C_{i+1}, soit des ponts en avant qui relient une clause de C'_i à l'extrémité inférieure de $C_\cup$.

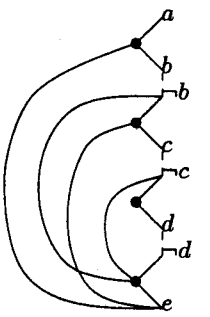


Figure 38: Configuration possible des ponts

De ce fait, la résolution linéaire L_i'' associée à C_i' est une résolution qui vérifie les conditions de la proposition 82 et ce, pour tout regroupement des dernières clauses de L_i''. Il est donc certain que chaque regroupement des dernières clauses de L_i'' fournira soit la même résolvante, soit une résolvante qui subsume celle de L_i''.

(a) $C_\cup$ contient au moins un pont tautologique

Le regroupement des clauses qui apparaissent sous ce pont tautologique forme une tautologie, puisque le cycle formé par le pont est élémentaire. Par regroupement, on peut donc transformer L_i en une résolution linéaire qui fournit le même résultat ou un résultat subsumant et qui fait intervenir une tautologie. R_{i+1} est donc soit une tautologie, soit une clause subsumée.

(b) $C_\cup$ ne contient aucun pont tautologique

i. $C_\cup$ contient au moins un pont en arrière

Tous les ponts en arrière partent du nœud clause de C_{i+1} Soit $\mathcal{P}$ le plus petit de ces ponts et l le littéral de ce pont.

Si l n'est pas l'extrémité supérieure de la chaîne, alors, L_{i+1}' produit une clause subsumée. En effet, on peut construire une autre résolution linéaire (non input) par regroupement des clauses sous $\mathcal{P}$ qui subsumera le résultat de L_{i+1}. En effet, le seul effet de ce regroupement est de faire disparaître l de la résolvante.

Si l est l'extrémité supérieure de la chaîne, il suffit de regrouper les clauses sous le pont pour obtenir une chaîne sans pont (les ponts en avant sont supprimés simultanément par fusion).

ii. $C_\cup$ ne contient aucun pont en arrière

Dans ce cas, tous les ponts sont des pont en avant qui relient une clause de C_i' à l'extrémité de la chaîne contenue dans C_{i+1}.

On peut supprimer tous les ponts de la chaîne par regroupement, en procédant du plus petit au plus grand des ponts.

Ce dernier théorème nous évite de considérer toutes les résolutions linéaires input associées à une chaîne.

7.2.6 Inutilité des cycles non élémentaires

La proposition suivante montre qu'on peut complètement se passer des cycles non élémentaires. Il s'agit d'un résultat crucial car il y a une infinité de cycles non élémentaires (pour peu qu'il existe au moins un cycle élémentaire).

Proposition 87:

Toute résolvante ni tautologique, ni subsumée, associée à une chaîne non ambiguë et non élémentaire, peut être obtenue par saturation en ne faisant intervenir que des chaînes élémentaires non ambiguës.

Démonstration 88:

Soit C une chaîne non ambiguë et non élémentaire. Soit n le premier nœud que l'on rencontre deux fois lorsqu'on parcourt la chaîne de gauche à droite. La chaîne peut donc s'écrire C_1–C_2–C_3 où C_2 est la première sous-chaîne de C d'extrémités n.

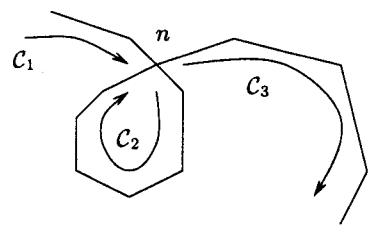


Figure 39: Premier nœud rencontré deux fois dans la chaîne non élémentaire

C_2 forme évidemment un cycle, qui est soit fusionnant, soit tautologique. Comme la chaîne C est non ambiguë, on peut regrouper les clauses des résolutions linéaires associées sans modifier la résolvante obtenue.

On regroupe donc les clauses de C_2. Si le cycle est tautologique, ce regroupement donnera une tautologie ou une clause subsumée et la résolvante associée à C sera également une tautologie ou une clause subsumée.

Si C_2 est un cycle fusionnant, on peut obtenir toute résolvante associée à C en regroupant d'abord les clauses de C_2, ce qui supprime ce cycle. On procède de même avec d'éventuels autres cycles de C. Comme ces regroupement sont nécessairement effectué dans la saturation, il est inutile de considérer des cycles non élémentaires.

Corollaire

Ce théorème permet de prouver de manière unifiée le théorème à un atome de liaison et celui à une clause de liaison. La démonstration est évidente pour ce dernier théorème (chaque cycle liant deux sous-bases passe deux fois par le nœud clause). Quant au théorème à un atome, soit chaque cycle liant deux sous-bases contient deux sous-chaînes C–l–C, soit il n'est pas élémentaire (il passe deux fois soit par l'atome, soit par sa négation).

7.2.7 Récapitulation des résultats

Les points à retenir des sections précédentes sont les suivants

- seules les résolvantes avec fusion sont utiles pour l'achèvement
- elles peuvent être produites par résolution linéaire input
- les résolvantes linéaires input avec fusion se traduisent sous la forme de cycles fusionnants
- les cycles fusionnants ambigus ou non élémentaires ne sont pas à considérer

On a vu également que les cycles intéressants ne sont pas tous indispensables. Ils peuvent générer des tautologies, des clauses subsumées, ou des clauses que le chaînage avant sait produire (cf la condition d'achèvement). Cependant, ces deux types de cycles seront faciles à éliminer. Il semble en revanche difficile d'aller au delà de la notion de cycle intéressant. On notera en particulier qu'il n'est pas possible de se contenter des cycles fondamentaux[7], ni de se passer de la saturation. En effet, il est nécessaire d'ajouter quatre clauses à la base cycle1-3 ci-dessous pour l'achever.

$$\left\{ \begin{array}{l} a_1 \vee a_2 \vee a_3 \\ \neg a_i \vee b_i \vee c_i \\ \neg c_i \vee d \end{array} \right\} i \in \{1,2,3\}$$

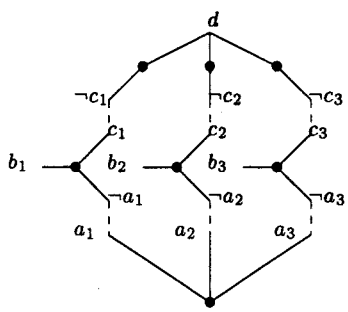


Figure 40: Graphe de la base cycle1-3

Si l'on n'effectue pas de saturation, il est impossible de produire la clause $b_1 \vee b_2 \vee b_3 \vee d$ pourtant indispensable. Si l'on se limite aux cycles fondamentaux, il manquera l'une des 3 clauses $b_i \vee b_j \vee a_k \vee d$ (i, j, k tous différents) selon le choix de la base de cycles. Cette clause manquante ne peut être produite ultérieurement par saturation (à moins, bien sûr, de changer de base de cycles).

7.2.8 Algorithme

On déduit des résultats précédents que l'algorithme ci-dessous calcule un achèvement d'une base $\mathcal{B}$. Dans cet algorithme, $\mathcal{B}_i$ contient la base après i étapes de saturation et C_i, les clauses ajoutées par l'étape de saturation numéro i.

1: $\mathcal{B}_0 \leftarrow \mathcal{B}$

2: $C_0 \leftarrow \mathcal{B}$

3: $i \leftarrow 0$

4:

5: **while** $C_i \neq \varnothing$ **do**

6: /* repérer les cycles intéressants */

7: Calculer S l'ensemble des clauses associées aux cycles intéressants de $\mathcal{B}_i$ dans lesquels intervient au moins une clause de C_i

8:

9: $C_{i+1} \leftarrow \varnothing$ /* éliminer les tautologies, les clauses subsumées et les conséquences qu'on peut obtenir de diverses manières sans que les fusions soient les mêmes */

[7]Un cycle fondamental est un cycle d'une base permettant de reconstruire tous les cycles.

10: **for all** clause C de S **do**
11: **if** C n'est ni tautologique, ni subsumée par une clause de $\mathcal{B}_i \cup \mathcal{C}_{i+1}$ **then**
12: **if** $\exists$ au moins une variante de C qui représente une déduction qui ne peut être effectuée par chaînage avant à partir de $\mathcal{B}_i \cup \mathcal{C}_{i+1}$ **then**
13: $\mathcal{C}_{i+1} \leftarrow \mathcal{C}_{i+1} \cup \{C\}$
14: **end if**
15: **end if**
16: **end for**
17:
18: $\mathcal{B}_{i+1} \leftarrow \mathcal{B}_i \cup \mathcal{C}_{i+1}$
19: $i \leftarrow i + 1$
20: **end while**

Il s'agit là d'une version naïve de l'algorithme car nous verrons dans la section suivante que les tests de subsomption, tautologie et celui utilisant le chaînage avant peuvent être incorporés à la recherche des cycles intéressants.

Sur la base cycle1-3, l'algorithme fonctionnera comme suit. Tout d'abord, $\mathcal{B}_0 = \mathcal{C}_0 = \mathcal{B}$. $\mathcal{B}_0$ contient trois cycles qui font tous intervenir des clauses de $\mathcal{C}_0$. Ces trois cycles sont fusionnants, élémentaires et non ambigus. Ils génèrent les clauses $a_1 \vee b_2 \vee b_3 \vee d$, $b_1 \vee a_2 \vee b_3 \vee d$ et $b_1 \vee b_2 \vee a_3 \vee d$. Comme ces clauses ne sont ni tautologiques, ni subsumées et ne peuvent s'obtenir sans fusion sur d, $\mathcal{C}_1$ les contient toutes et elles sont toutes ajoutées au graphe. Suite à cet ajout, $\mathcal{B}_1$ contient trois cycles fusionnants, élémentaires et non ambigus qui font intervenir des clauses de $\mathcal{C}_1$. Ces trois cycles génèrent la même clause $b_1 \vee b_2 \vee b_3 \vee d$ qui n'est ni tautologique, ni subsumée et ne peut être produite sans fusion sur d. On a donc $\mathcal{C}_2 = \{b_1 \vee b_2 \vee b_3 \vee d\}$ que l'on ajoute à $\mathcal{B}_1$ pour obtenir $\mathcal{B}_2$. Cette dernière clause ajoutée ne génère plus de cycle fusionnant dans le graphe. L'algorithme s'arrête avec $\mathcal{C}_3 = \varnothing$. Si l'on souhaite obtenir une base de règles, la base $\mathcal{B}$ est achevée par l'ajout aux variantes des clauses de $\mathcal{B}$, des variantes des quatre clauses citées ayant d comme conclusion (puisque c'est à chaque fois le littéral fusionné).

7.2.9 La simplification par chaînage avant

Comme annoncé précédemment, les tests de tautologie, subsomption et simplification par chaînage avant peuvent être incorporés à la recherche des cycles, ce qui permet d'élaguer au plus tôt l'arbre de recherche.

On notera d'abord qu'on a tout intérêt à rechercher les cycles intéressants en partant de leur tête, puisque cela nous assure d'avoir au moins une fusion dans la résolvante. Chaque fois que l'on ajoute une clause au cycle en cours de construction, on met à jours la liste L des «piquants» rencontrés jusque là. Cette liste représente une partie de la résolvante finale. Si cette liste est tautologique ou subsumée par une autre clause de la base, il est évidemment inutile de poursuivre la construction. Il reste alors à effectuer le test par chaînage avant. On remarque d'abord que, si t est la tête du cycle, la règle correspondant au cycle est de la forme $(\neg L) \wedge \ldots \rightarrow t$. On calcule alors ce que produirait le chaînage avant à partir des faits $\neg L$. Si $t \in Fwch(\mathcal{B} \cup \neg L)$ on peut arrêter immédiatement la construction du cycle puisque la clause qu'on produirait serait redondante. Mais on peut aller plus loin. Si le chaînage avant produit un littéral l, alors il est inutile de prolonger le cycle en effectuant une résolution sur l. En effet, $l \in Fwch(\mathcal{B} \cup \neg L)$ signifie que $L \vee l$ est un impliqué premier qu'on peut obtenir par une résolution linéaire input sans fusion sur l ni sur un

littéral servant de pivot. Or, le reste du cycle ne contient pas de fusion sur l, ou s'il en contient, il y aura un autre cycle pour représenter cette clause. Si donc l'on poursuit le cycle, on obtiendra une résolvante qu'il est possible de produire par une résolution linéaire input sans fusion sur l ni sur un littéral servant de pivot. Cette résolvante sera donc redondante. Il est également inutile de prolonger le cycle en effectuant une résolution sur $\neg l$ car on peut effectuer le même raisonnement en parcourant le cycle dans l'autre sens. Donc, on peut éviter de prolonger le cycle en cours de construction en passant par un atome correspondant à un littéral de $Fwch(\mathcal{B} \cup \neg L)$.

Exemple 89:

Soit la base

$$\begin{cases} a \vee b \vee c \\ \neg c \vee d \vee e \\ \neg e \vee f \\ \neg f \vee a \\ b \vee d \vee e \end{cases}$$

Son graphe de littéraux est le suivant

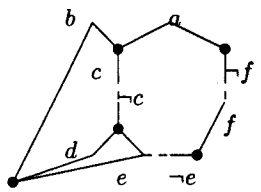


Figure 41: Exemple de simplification par chaînage avant incorporée à la recherche de cycle

Les quatre premières clauses de la base forment un cycle qui génère la clause $b \vee d \vee \underline{a}$. Cependant, il est inutile de l'ajouter puisque cette clause s'obtient par résolution linéaire input à partir des trois dernières clauses et ce, sans fusion sur a. Voici comment on le détecte.

On commence la recherche de cycle à partir du point a qui est un littéral fusionné potentiel puisqu'il appartient à deux clauses. Si l'on ajoute alors au cycle en construction les clauses $a \vee b \vee c$ et $\neg c \vee d \vee e$, l'on sait déjà que b et d feront partie de la clause générée. On ajoute alors leur négation à la base de fait et on lance le chaînage avant qui produit e. Cela montre que de quelque manière que l'on boucle le cycle, la clause générée sera inutile puisque l'on pourra l'obtenir par une résolution linéaire input sans fusion sur a. De ce fait, on évite les résolutions avec $\neg e \vee f$ et $\neg f \vee a$.

7.2.10 Utilisation pour le recueil de connaissance

L'achèvement par cycles peut être particulièrement intéressant lors du recueil de connaissances. Par exemple, un expert pourrait saisir ses règles dans un éditeur qui effectue un achèvement incrémental et signale toute règle conséquence de la connaissance introduite et qu'il faut ajouter pour garantir la complétude du chaînage avant. Par la même, l'expert peut vérifier la cohérence de ses règles, puisque le système fournit une explication de l'origine de la règle. Toute règle dérivée suspecte permet donc d'identifier les règles de validité douteuse dont elle découle. Un exemple de session avec un tel éditeur est donné ci-dessous. L'expert entre d'abord 4 règles. La dernière permet avec deux règles précédentes de déduire

$d \rightarrow e$ ce que ne sait faire le chaînage avant. Le système le signale, et dans notre exemple, l'expert accepte l'inclusion automatique de la règle déduite.

```
1> not a -> not d or b
2> c -> d
3> a -> e
4> b -> e
Les règles 1,3 et 4 impliquent d -> e ce que le chaînage
avant ne sait pas déduire. Faut-il ajouter cette règle ? o
5> déduit de 1,3 et 4 : d -> e
```

7.2.11 Forces et faiblesses de la méthode

Nous précisons maintenant quelques détails sur les restrictions qui font l'intérêt de la méthode ainsi que ses faiblesses.

D'abord, la recherche des cycles est menée à partir de l'extrémité du cycle, c'est à dire, à partir des éventuels littéraux fusionnés. De ce fait, on évite toute résolution si aucune fusion n'existe. Ce n'est pas le cas dans les approches où les résolvantes avec fusions sont filtrées après production par un algorithme d'énumération d'impliqués premiers [MD94b][dV94].

Ensuite, l'approche par cycles interdit d'effectuer des résolutions entre des clauses appartenant à des composantes 2-connexes différentes. On retrouve donc les simplifications de l'achèvement par parties et l'efficacité qu'on lui connaît.

Par ailleurs, le test de simplification par chaînage avant est directement incorporé à la recherche de cycles ce qui permet d'élaguer au plus tôt l'arbre de recherche. On a alors la garantie d'obtenir une base irredondante.

L'algorithme est également incrémental. En effet, la recherche de cycles peut s'effectuer à chaque fois que l'on ajoute une clause. On se restreint alors aux seuls cycles contenant cette clause.

Enfin, avant de chercher à prolonger un chemin par une clause, il est souhaitable de vérifier que le chemin que l'on obtiendra pourra permettre de clore le cycle. Ceci peut se faire en conservant une information sur la clôture transitive de la relation $\mathcal{R}$ définie par $C\mathcal{R}C'$ s'il existe une chaîne $C{-}l{-}\cdots{-}\neg l{-}C'$. Toutefois, cette optimisation ne semble pas très facile à programmer.

En fait, le défaut de la méthode est de ne pas mener à une implantation simple malgré les restrictions qu'elle comporte implicitement. Une bonne implantation devrait effectuer des résolutions uniquement si elles peuvent boucler un cycle. Les résolvantes partielles obtenues devraient être stockées dans un cache en vue d'une réutilisation dans un autre cycle. Le point le plus délicat semble être la gestion de ce cache, tout particulièrement lors de l'ajout d'une nouvelle clause.

7.3 Exemples d'achèvement de complexité exponentielle

Le premier cas d'achèvement exponentiel identifié est celui de la base pigeon-n-m [MD94b]. Malheureusement, le graphe de cette base est déjà trop complexe pour pouvoir l'analyser simplement comme on peut le constater ci-dessous.

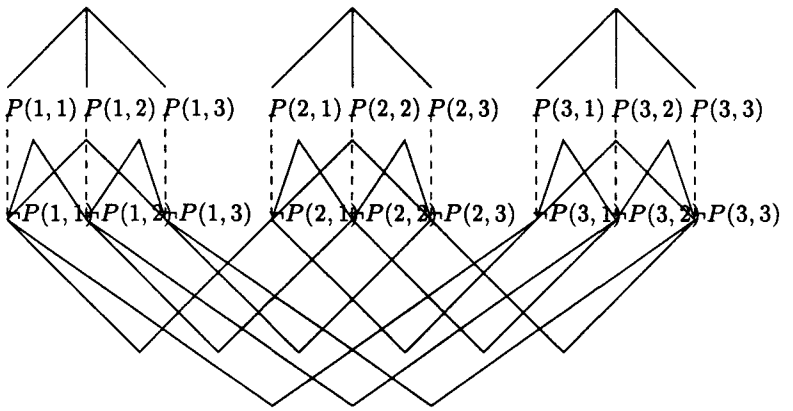
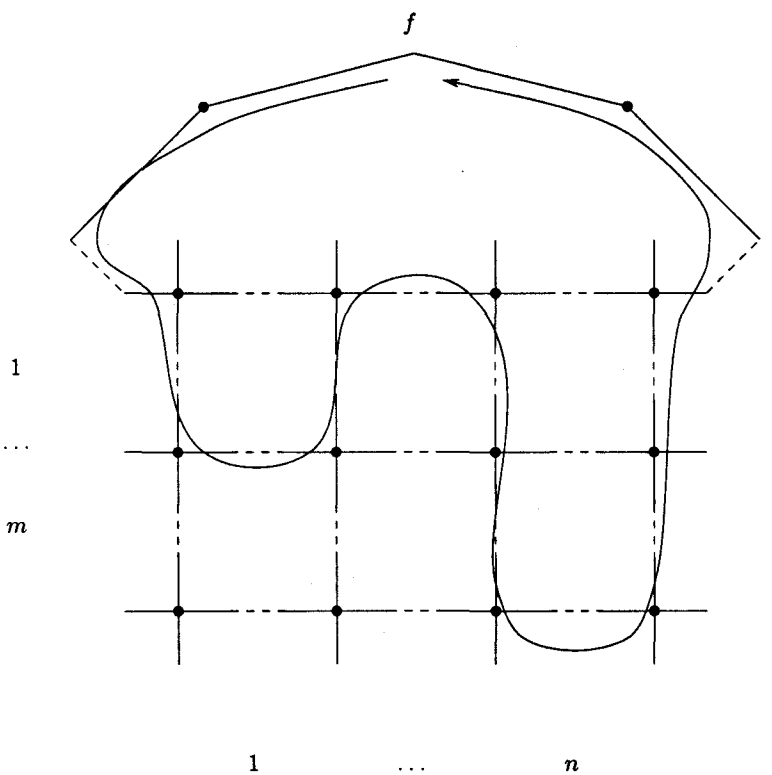
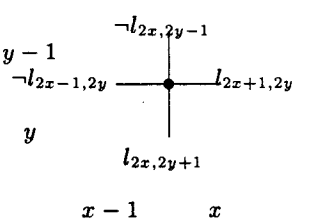


Figure 42: Graphe de la base pigeon-3-3

Une base beaucoup plus simple et dont l'achèvement est exponentiel est la base cycle4-n-m. Sa définition précise figure en annexe. Le graphe de cette base est constitué d'un carroyage dont les chemins sont des chaînes de résolutions. Deux clauses sont ajoutées à ce quadrillage pour que tous les chemins du carroyage reliant deux points particuliers puissent être étendus pour former un cycle intéressant. De plus, aucune subsomption ne peut s'appliquer entre de tels cycles. La figure ci-dessous représente le graphe de cette base ainsi qu'un exemple de cycle intéressant.





Convention de numérotation

Figure 43: Graphe de cycle4-n-m

Il est clair que l'on peut construire des cycles du graphe en parcourant chacune des n cases en abscisse à l'une des m ordonnées possibles. Il y a donc au moins m^n cycles intéressants. Comme il n'y a aucune subsomption possible (il y a bijection entre les éléments de

la résolvante et le chemin parcouru dans le graphe), il faut ajouter au moins m^n clauses pour l'achever. On notera que ce nombre est une borne inférieure puisque l'on peut construire d'autres cycles qui repassent plus d'une fois à une abscisse donnée. De plus, aucune saturation n'est nécessaire pour obtenir cette explosion combinatoire.

Un dernier exemple de base dont l'achèvement est exponentiel est celui de la base cycle1-n dont le graphe est le suivant (sa définition figure en annexe)

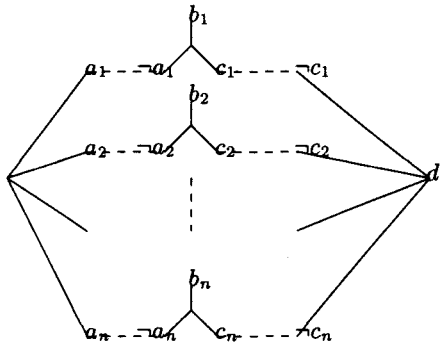


Figure 44: Graphe de la base cycle1-n

Soit C_i la clause $\neg a_i \vee b_i \vee c_i$ et C la clause $a_1 \vee a_2 \vee \ldots \vee a_n \vee d$ qui ne figure pas sur le graphe. Lors de la première étape de saturation, il est possible de former un cycle en passant par deux clauses C_i et C_j ($i \neq j$). Il y a donc $n(n-1)/2$ cycles qui génèrent les clauses obtenues à partir de C en remplaçant deux a_i différents par un b_i d'indice correspondant (par exemple, en passant par C_2 et C_4, la clause obtenue est $a_1 \vee b_2 \vee a_3 \vee b_4 \vee a_5 \vee a_6 \vee \ldots \vee a_n \vee d$). Lors de la deuxième étape de saturation, les clauses générées précédemment forment avec l'une des clauses C_i de nouveaux cycles. Ces cycles génèrent les clauses obtenues à partir de C en remplaçant trois a_i différents par un b_i d'indice correspondant. L'étape suivante de saturation génère les clauses obtenues à partir de C en remplaçant quatre a_i différents par un b_i d'indice correspondant et ainsi de suite jusqu'à ce que tous les a_i de C soient remplacés par des b_i. On peut énumérer facilement les clauses ajoutées. Il y en a

$$\sum_{i=0}^{n-2} C_n^i = 2^n - n - 1$$

Elles sont toutes de longueur $n+1$. Il s'agit donc bien d'un cas d'achèvement exponentiel.

Chapitre 8

Comparaison avec des méthodes complètes

Une alternative à l'emploi de l'achèvement est d'utiliser une méthode complète de production sur la base initiale. Nous comparons donc l'usage de l'achèvement avec une telle méthode qui peut être

- soit une méthode de production basée sur un test de satisfiabilité
- soit une méthode basée sur les champs de production de [Sie87]

8.1 Comparaison avec une méthode basée sur SAT

On peut construire un algorithme complet de production de littéraux à partir d'un test de satisfiabilité comme suit :

1: **if** $\mathcal{B} \cup F$ insatisfiable **then**
2: return *Insatisfiable*
3: **end if**
4:
5: **for all** littéral $l \in Lit(\mathcal{B})$ **do**
6: **if** $l \in F$ **then**
7: continue ; /* l est trivialement impliqué */
8: **end if**
9: **if** $\neg l \in F$ **then**
10: continue ; /* l ne peut être impliqué puisque la base est satisfiable */
11: **end if**
12: **if** $\mathcal{B} \cup F \cup \{\neg l\}$ insatisfiable **then**
13: $F \leftarrow F \cup \{l\}$ /* l est impliqué */
14: **end if**
15: **end for**
16: return F

On compare alors les performances de cet algorithme sur la base $\mathcal{B}$ par rapport à celles du chaînage avant sur la base $Achvt(\mathcal{B})$. On effectue ce test sur des bases de faits tirées aléatoirement et de longueur variant entre 1 et le nombre d'atomes de la base. Les mesures que l'on effectue sont les suivantes.

- n
  nombre de bases aléatoires de faits testées. Les longueurs de ces bases de faits varient entre 1 et le nombre d'atome de la base. Il y a par ailleurs autant de bases de longueur n que de bases de longueur $m \neq n$.
- t_{sat}
  temps nécessaire à l'algorithme basé sur le test de satisfiabilité pour effectuer les inférences sur les n bases de faits. Le test de satisfiabilité utilisé a des performances équivalentes à C-SAT [DABC94].
- t_{chavt}
  temps mis par le chaînage avant pour effectuer les mêmes inférences à partir de la base achevée.
- t_{comp}
  temps nécessaire à l'achèvement de la base

On déduit de ces temps deux rapports. Le premier mesure l'accélération moyenne G_{sat} sur chaque inférence que procure l'usage du chaînage avant sur $Achvt(\mathcal{B})$. Il est défini par

$$G_{sat} = \frac{t_{sat}}{t_{chavt}}$$

Le second rapport détermine le nombre moyen d'inférences R_{sat} nécessaires pour rentabiliser le temps de compilation. Il se déduit de l'équation

$$R_{sat} \cdot \frac{(t_{sat} - t_{chavt})}{n} = t_{comp}$$

et vaut

$$R_{sat} = \frac{n.t_{comp}}{(t_{sat} - t_{chavt})}$$

Le tableau suivant présente les résultats expérimentaux pour quelques bases de connaissances.

Nom	n_{at}	n_{cl}	t_{comp}	t_{sat}	t_{chavt}	n	G_{sat}	R_{sat}
deputes	16	20	0.425	0.707	0.216	450	3.2	389.6
pannes	16	30	0.255	0.737	0.196	450	3.7	212.0
logiciens	11	15	0.115	0.378	0.136	300	2.7	142.3
type1-76	76	75	1.99	18.268	2.246	1820	8.1	226.0
type3-16	33	17	0.365	2.726	0.611	960	4.4	165.6
type5-10	41	31	0.305	4.314	0.860	1200	5.0	105.9
type6-11	45	44	0.355	4.992	1.036	1320	4.8	118.4
type7-5	26	20	0.135	1.671	0.375	750	4.4	78.0
type7-6	31	24	0.185	2.433	0.494	900	4.9	85.8
pigeon-2-3	6	11	0.025	0.114	0.045	150	2.5	53.7
pigeon-3-4	12	33	0.265	0.477	0.176	330	2.7	290.6

.../...

.../...

Nom	n_{at}	n_{cl}	t_{comp}	t_{sat}	t_{chavt}	n	G_{sat}	R_{sat}
pigeon-4-5	20	74	19.6	1.324	1.138	570	1.1	60109.8
ramsey-4	18	36	41.5	1.061	0.427	510	2.4	33389.3
nqueens-6	36	296	10.5	8.88	0.541	1050	16.4	1322.1
chandra21	24	21	55.2	1.597	1.253	690	1.2	110716
chandra24	27	24	442	2.089	1.772	780	1.1	1087850
easy	9	26	0.085	0.282	0.077	240	3.6	99.4
ex1	6	14	0.025	0.123	0.041	150	2.9	45.8
ex2	9	24	0.045	0.260	0.080	240	3.2	60.0
ex3	21	106	28.8	2.080	0.553	600	3.7	11316.7
ex4	9	26	0.085	0.277	0.080	240	3.4	103.6
history-ex	21	17	0.095	1.016	0.283	600	3.5	77.7
selenoid	11	19	0.115	0.376	0.100	300	3.7	125.3
valve	13	50	0.275	0.796	0.139	360	5.7	150.5
two-pipes	15	54	1.75	0.849	0.212	420	3.9	1154.3
three-pipes	21	82	19.7	1.724	0.433	600	3.9	9155.5
four-pipes	27	110	127	2.856	0.739	780	3.8	46799.2
mine-2-2	52	2088	1.57	61.937	1.099	972	56.3	25.0
mine-3-2	74	3128	20.6	122.49	1.037	525	118.0	89.0
adder-5	21	71	0.125	1.967	0.293	600	6.7	44.7
adder-10	41	141	0.325	9.714	0.819	1124	11.8	41.0
adder-20	81	281	0.935	355.363	0.715	561	496.5	1.4
adder-25	101	351	1.58	374.657	0.472	305	792.2	1.2
mult-3-3	31	304	326	5.107	1.612	900	3.1	83966.4
mult-inf-3-3	31	311	121	5.090	1.078	900	4.7	27145
mult-sup-3-3	31	311	65.7	5.188	0.901	900	5.7	13791.5
2tree-11	22	27	6.26	1.313	0.309	630	4.2	3929.2
cycle1-6	19	13	5.99	0.874	0.322	540	2.7	5861.9

Dans la plupart des cas, l'usage du chaînage avant permet d'être au moins 2 à 3 fois plus rapide que la méthode basée sur le test de satisfiabilité. Il s'agit là d'une mesure moyenne sur toutes les bases de faits. En fait, l'algorithme basé sur SAT s'exécute d'autant plus vite qu'il y a de faits (la base est davantage simplifiée) comme on peut le voir sur la figure 45. Le chaînage avant est quant à lui légèrement plus lent quand il y a plus de faits.

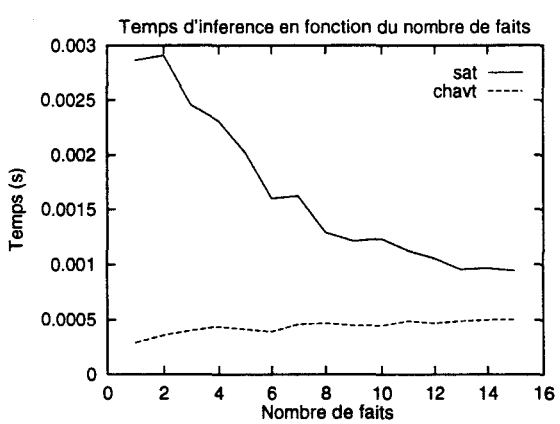


Figure 45: Comparaison entre les temps d'inférence pour le chaînage avant et l'algorithme basé sur SAT en fonction du nombre de faits pour la base pannes

L'achèvement est donc une technique rentable dans la plupart des cas. On peut de plus constater que le seuil de rentabilité reste relativement bas. Il est assez souvent de l'ordre de 100 à 1000 et ne dépasse pas 10^6 dans le pire des cas. Ce dernier chiffre peut sembler énorme. En fait, il ne l'est pas vraiment puisque l'achèvement est une méthode destinée à des bases qui sont fréquemment interrogées telles que les systèmes de détection de pannes. De plus, ce seuil de rentabilité dépend très fortement des optimisations de la compilation et pourrait donc s'écrouler en utilisant une implémentation réellement optimisée. Enfin, un seuil de rentabilité élevé ne signifie pas un gain G faible comme on peut le voir sur les bases mult-*-*.

On peut comprendre très simplement pourquoi l'usage du chaînage avant est plus efficace que celui d'un test de satisfiabilité. D'abord, le chaînage avant est un algorithme plus simple que le test de satisfiabilité. De ce fait, l'exécution de sa boucle principale est plus rapide que dans le cas du test SAT. Comparé à un chaînage avant, la méthode basée sur le test de satisfiabilité ne sait pas non plus réutiliser au plus tôt un littéral prouvé et il faut effectuer de multiples tests pour obtenir un algorithme de production. De plus, le test de satisfiabilité est essentiellement un algorithme de Davis et Putnam. Or, on peut considérer que le chaînage avant est une simplification de l'algorithme de Davis et Putnam où le littéral choisi à chaque étape ne peut être qu'un littéral figurant dans une clause unitaire (ce qui le rend bien sûr incomplet). Le test de satisfiabilité peut donc être vu comme un chaînage avant augmenté de la construction d'un arbre binaire. On comprend donc bien que lorsque la taille de la base achevée ne dépasse pas de manière excessive celle de la base initiale, le chaînage avant est gagnant. En fait, les seules bases pour lesquelles l'achèvement n'est pas rentable sont des bases dont l'achevé est exponentiel et qu'on peut supposer ne pas représenter des problèmes courants.

Enfin, il faut noter que le chaînage avant peut être directement transcrit en langage d'assemblage (même si cela n'a pas été effectué pour ces mesures). Il suffit en effet de transcrire chaque règle en un *if...then...* ce qui permet d'obtenir une efficacité maximale. Un telle approche n'est guère possible avec un test de satisfiabilité car chaque clause doit être traitée à un niveau méta-interprète.

8.2 Comparaison avec une méthode basée sur les champs de production

Une méthode naturelle de production des littéraux impliqués est celle des champs de production [Sie87]. Comme nous souhaitons connaître tous les littéraux impliqués par la base et les faits, le champ que nous considérons est l'ensemble des clauses unitaires. Un algorithme sémantique qui permet d'effectuer ce calcul est le suivant :

1: /* simplifier la base */
2: $H = Fwch(\mathcal{B} \cup F)$
3: **for all** littéral $l \in H$ **do**
4: $\mathcal{B}$=Simplifier($\mathcal{B}$,l)
5: **end for**
6:
7: /* calculer les littéraux impliqués
la méthode employée revient à calculer un arbre sémantique clausal à partir des modèles de $\mathcal{B}$ et à rejeter toute clause de longueur supérieure ou égale à deux */
8: calculer l'intersection I des modèles de $\mathcal{B}$ produits par la procédure de Davis et Putnam
9:

10: return $H \cup I$

Une optimisation a été incorporée à cet algorithme pour éviter l'énumération de certaines interprétations qui ne peuvent modifier l'intersection en cours de calcul. On mesure les temps de calcul de cet algorithme t_{prod} dans les mêmes conditions que pour l'algorithme basé sur SAT. Les temps d'inférences ont été limités à une minute par base de faits et tout dépassement est signalé par un tiret. Les résultats sont les suivants :

Nom	n_{at}	n_{cl}	t_{achvt}	t_{prod}	t_{chavt}	n	G_{prod}	R_{prod}
deputes	16	20	0.425	0.567	0.216	450	2.6	545.1
pannes	16	30	0.255	0.575	0.196	450	2.9	302.4
logiciens	11	15	0.115	0.289	0.136	300	2.1	225.1
type1-76	76	75	1.99	9.626	2.246	1820	4.2	490.7
type3-16	33	17	0.365	2.114	0.611	960	3.4	233.2
type5-10	41	31	0.305	5.281	0.860	1200	6.1	82.7
type6-11	45	44	0.355	4.446	1.036	1320	4.2	137.4
type7-5	26	20	0.135	1.330	0.375	750	3.5	105.9
type7-6	31	24	0.185	1.946	0.494	900	3.9	114.7
pigeon-2-3	6	11	0.025	0.088	0.045	150	1.9	85.3
pigeon-3-4	12	33	0.265	0.370	0.176	330	2.1	450.6
pigeon-4-5	20	74	19.6	1.029	1.138	570	0.9	-102165
ramsey-4	18	36	41.5	0.798	0.427	510	1.8	56966.8
nqueens-6	36	296	10.5	6.868	0.541	1050	12.6	1742.4
chandra21	24	21	55.2	1.164	1.253	690	0.9	-427638
chandra24	27	24	442	1.508	1.772	780	0.8	-1308180
easy	9	26	0.085	0.222	0.077	240	2.8	140.1
ex1	6	14	0.025	0.098	0.041	150	2.3	65.8
ex2	9	24	0.045	0.208	0.080	240	2.5	84.3
ex3	21	106	28.8	1.603	0.553	600	2.8	16450.9
ex4	9	26	0.085	0.222	0.080	240	2.7	143.4
history-ex	21	17	0.095	0.803	0.283	600	2.8	109.5
selenoid	11	19	0.115	0.301	0.100	300	2.9	171.7
valve	13	50	0.275	0.673	0.139	360	4.8	185.2
two-pipes	15	54	1.75	0.671	0.212	420	3.1	1601.7
three-pipes	21	82	19.7	1.361	0.433	600	3.1	12741.2
four-pipes	27	110	127	2.289	0.739	780	3.0	63935.6
mine-2-2	52	2088	1.57	10.797	1.099	972	9.8	157.3
mine-3-2	74	3128	20.6	11.471	1.037	525	11.0	1036.6
adder-5	21	71	0.125	1.549	0.293	600	5.2	59.7
adder-10	41	141	0.325	126.995	0.819	1124	154.9	2.8
adder-20	81	281	0.935	-	0.715	561	-	-
adder-25	101	351	1.58	-	0.472	305	-	-
mult-3-3	31	304	326	4.102	1.612	900	2.5	117850
mult-inf-3-3	31	311	121	4.085	1.078	900	3.7	36212.9
mult-sup-3-3	31	311	65.7	4.090	0.901	900	4.5	18543.2
2tree-11	22	27	6.26	1.053	0.309	630	3.4	5299.2
cycle1-6	19	13	5.99	0.671	0.322	540	2.0	9280.1

On remarque que l'achèvement n'est pas rentable pour trois des bases du jeu de test (pigeon-4-5, chandra21 et chandra24) puisque l'algorithme de production est plus rapide que le chaînage avant sur la base achevée. Cela s'explique par le fait que dans ces trois cas, la base achevée est de taille exponentielle par rapport à la base initiale.

Il est intéressant de comparer les gains et rentabilité des deux algorithmes.

Nom	n_{at}	n_{cl}	G_{sat}	G_{prod}	R_{sat}	R_{prod}
deputes	16	20	3.2	2.6	389.6	545.1
pannes	16	30	3.7	2.9	212.0	302.4
logiciens	11	15	2.7	2.1	142.3	225.1
type1-76	76	75	8.1	4.2	226.0	490.7
type3-16	33	17	4.4	3.4	165.6	233.2
type5-10	41	31	5.0	6.1	105.9	82.7
type6-11	45	44	4.8	4.2	118.4	137.4
type7-5	26	20	4.4	3.5	78.0	105.9
type7-6	31	24	4.9	3.9	85.8	114.7
pigeon-2-3	6	11	2.5	1.9	53.7	85.3
pigeon-3-4	12	33	2.7	2.1	290.6	450.6
pigeon-4-5	20	74	1.1	0.9	60109.8	-102165
ramsey-4	18	36	2.4	1.8	33389.3	56966.8
nqueens-6	36	296	16.4	12.6	1322.1	1742.4
chandra21	24	21	1.2	0.9	110716	-427638
chandra24	27	24	1.1	0.8	1.0e+06	-1.3e+06
easy	9	26	3.6	2.8	99.4	140.1
ex1	6	14	2.9	2.3	45.8	65.8
ex2	9	24	3.2	2.5	60.0	84.3
ex3	21	106	3.7	2.8	11316.7	16450.9
ex4	9	26	3.4	2.7	103.6	143.4
history-ex	21	17	3.5	2.8	77.7	109.5
selenoid	11	19	3.7	2.9	125.3	171.7
valve	13	50	5.7	4.8	150.5	185.2
two-pipes	15	54	3.9	3.1	1154.3	1601.7
three-pipes	21	82	3.9	3.1	9155.5	12741.2
four-pipes	27	110	3.8	3.0	46799.2	63935.6
mine-2-2	52	2088	56.3	9.8	25.0	157.3
mine-3-2	74	3128	118.0	11.0	89.0	1036.6
adder-5	21	71	6.7	5.2	44.7	59.7
adder-10	41	141	11.8	154.9	41.0	2.8
adder-20	81	281	496.5	-	1.4	-
adder-25	101	351	792.2	-	1.2	-
mult-3-3	31	304	3.1	2.5	83966.4	117850
mult-inf-3-3	31	311	4.7	3.7	27145	36212.9
mult-sup-3-3	31	311	5.7	4.5	13791.5	18543.2
2tree-11	22	27	4.2	3.4	3929.2	5299.2
cycle1-6	19	13	2.7	2.0	5861.9	9280.1

On constate alors que l'algorithme de production sémantique est environ 30% plus rapide que celui basé sur le test de satisfiabilité sauf dans les cas où la base a de nombreux modèles que l'optimisation ne permet pas d'ignorer (cas de l'additionneur).

Troisième partie

L'achèvement en calcul des prédicats

Chapitre 9

Définitions et notations

9.1 Les langages du premier ordre

Nous rappelons ici brièvement les notions de logique du premier ordre que nous utilisons et précisons les notations. Des présentations plus complètes peuvent être trouvées dans [CL73] ou [Lov78].

Un langage est d'abord défini par son alphabet.

Définition 90:

Un **langage du premier ordre** est déterminé par la donnée de cinq ensembles :

- un ensemble $\mathcal{Q} = \{\forall, \exists, \vee, \wedge, \neg, \rightarrow, \leftrightarrow, \ldots\}$ de **connecteurs** et **quantificateurs**.
- un ensemble $\mathcal{P} = \{P, Q, R, \ldots\}$ de **symboles de prédicats**. À chaque symbole de prédicat est associé un nombre entier appelé **arité** du symbole et précisant combien le prédicat a d'arguments. On choisit de représenter un prédicat par une lettre majuscule suivie de la liste de ses arguments entre parenthèses : $P(X_1, \ldots, X_n)$. On note $\boldsymbol{P/n}$ pour indiquer que le prédicat P est d'arité n.
- un ensemble $\mathcal{F} = \{f, g, h, \ldots\}$ de **symboles de fonctions**. À chaque symbole de fonction est associé un nombre entier non nul appelé **arité** du symbole et précisant combien la fonction a d'arguments. On choisit de représenter une fonction par une lettre minuscule suivie de la liste de ses arguments entre parenthèses : $f(X_1, \ldots, X_n)$. On note $\boldsymbol{f/n}$ pour indiquer que la fonction f est d'arité n.
- un ensemble $\mathcal{V} = \{X, Y, Z, \ldots\}$ de **variables**. On choisit de représenter les variables par des lettres majuscules.
- un ensemble $\mathcal{C} = \{a, b, c, \ldots\}$ de **constantes**. On choisit de représenter les constantes par des lettres minuscules.

Les règles de grammaire d'un langage du premier ordre sont les suivantes :

Définition 91:

Un **terme** est

- soit une constante
- soit une variable
- soit une expression de la forme $f(t_1, \ldots, t_n)$ où f est un symbole de fonction d'arité n et t_1 à t_n des termes.

La **profondeur d'un terme** est égale à

- 1 si le terme est une constante ou une variable
- 1 plus la plus grande des profondeurs de $t_1, \ldots, t_n$ si le terme est de la forme $f(t_1, \ldots, t_n)$.

Un terme est **clos** s'il ne contient pas de symbole de variable.

Définition 92:

Un **atome** est une expression de la forme $P(t_1, \ldots, t_n)$ où P est un symbole de prédicat d'arité n et t_1 à t_n des termes.

Un **littéral** est un atome $P(t_1, \ldots, t_n)$ ou sa négation $\neg P(t_1, \ldots, t_n)$.

On note également $|l|$ l'atome correspondant au littéral l.

Définition 93:

Une **formule** est

- soit un littéral
- soit l'une des formules $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$ à condition que F et G soient des formules.
- soit l'une des formules $\forall X, (F)$ ou $\exists X, (F)$ à condition que F soit une formule dans laquelle n'apparaissent ni $\forall X$ ni $\exists X$. On dit que F est dans la **portée du quantificateur**.

Une formule doit être de taille finie.

Une variable est **libre** si elle n'est dans la portée d'aucun quantificateur la concernant. Autrement, elle est **liée**.

Définition 94:

On appelle **vocabulaire** d'une formule F le quintuplet $\boldsymbol{Voc(F)}$ formé par l'ensemble des connecteurs, symboles de constantes, symboles de variables, symboles de fonctions et symboles de prédicats qui figurent dans F.

Définition 95:

Une formule est en forme **prénexe** si tous les quantificateurs universels et existentiels figurent en début de clause avant tout littéral.

Définition 96:

Une **clause** est une disjonction de littéraux sous forme prénexe quantifiée universellement.

Définition 97:

Une **substitution** $\sigma = \{X_1 \rightarrow t_1, \ldots, X_n \rightarrow t_n\}$ est une application à support fini qui à une formule F associe la formule $\sigma(F)$ obtenue en remplaçant simultanément toutes les occurrences des variables X_i de F par le terme t_i correspondant. $\sigma(F)$ est appelé **instance** de F. On note $\boldsymbol{id}$ la substitution identité.

Définition 98:

Une **variante alphabétique** d'une formule f est une formule f' obtenue par renommage des variables de f. Plus précisément, f' est une variante alphabétique de f si et seulement si il existe deux substitutions σ_1 et σ_2 telles que $\sigma_1(f) = f'$ et $f = \sigma_2(f')$.

Définition 99:

Une **interprétation** I d'un langage du premier ordre $\mathcal{L} = (\mathcal{Q}, \mathcal{P}, \mathcal{F}, \mathcal{V}, \mathcal{C})$ est une application de l'ensemble des formules dans $\{\mathbb{V}, \mathbb{F}\}$.

Elle est définie par la donnée

- d'un ensemble D appelé **domaine**
- d'une application ϕ qui
  - à chaque constante de $\mathcal{C}$ associe un élément de D
  - à chaque fonction d'arité n de $\mathcal{F}$ fait correspondre une fonction de D^n dans D
  - à chaque prédicat d'arité n de $\mathcal{P}$ fait correspondre une fonction de D^n dans $\{\mathbb{V}, \mathbb{F}\}$

ϕ est étendue canoniquement aux termes et prédicats :

- si f est une fonction d'arité n, $\phi(f(t_1, \ldots, t_n)) = \phi(f)(\phi(t_1), \ldots, \phi(t_n))$
- si P est un prédicat d'arité n, $\phi(P(t_1, \ldots, t_n)) = \phi(P)(\phi(t_1), \ldots, \phi(t_n))$

Une **valuation** $V = \{X_1 \to d_1, \ldots, X_n \to d_n\}$ est une application à support fini qui à une formule F associe la formule $V(F)$ obtenue en remplaçant simultanément toutes les occurrences des variables X_i de F par l'élément d_i de D correspondant.

L'interprétation d'une formule découle des règles suivantes :

- Si P est un prédicat d'arité n alors, $I(P(t_1, \ldots, t_n)) = \phi(P(t_1, \ldots, t_n))$
- $I(\neg f) = \mathbb{V} \Leftrightarrow I(f) = \mathbb{F}$
- $I(\forall X,\ f) = \mathbb{V}$ si pour tout élément $d \in D$, la valuation $V = \{X \to d\}$ est telle que $I(V(f)) = \mathbb{V}$.
- $I(\exists X,\ f) = \mathbb{V}$ si il existe au moins un élément $d \in D$ et la valuation $V = \{X \to d\}$ tels que $I(V(f)) = \mathbb{V}$.
- $I(f_1 \wedge f_2) = \mathbb{V} \Leftrightarrow I(f_1) = I(f_2) = \mathbb{V}$
- $I(f_1 \vee f_2) = \mathbb{F} \Leftrightarrow I(f_1) = I(f_2) = \mathbb{F}$
- $I(f_1 \to f_2) = \mathbb{F} \Leftrightarrow I(f_1) = \mathbb{F}$ et $I(f_2) = \mathbb{V}$
- $I(f_1 \leftrightarrow f_2) = \mathbb{V} \Leftrightarrow I(f_1) = I(f_2)$

Une formule qui contient des variables libres ne peut être interprétée. Toute formule sans variable libre qui n'est pas interprétée à $\mathbb{V}$ est interprétée à $\mathbb{F}$ et réciproquement.

Pour pouvoir interpréter toute formule, on considère que les variables libres d'une formule sont implicitement quantifiées universellement en tête de formule

Définition 100:

Un ensemble de termes ou de formules $t_1, \ldots, t_n$ est **unifiable** s'il existe une substitution σ telle que $\sigma(t_1) = \ldots = \sigma(t_n)$. Dans ce cas, σ est un **unificateur** de $t_1, \ldots, t_n$.

Une substitution σ est un **unificateur le plus général** (**m.g.u.** en anglais) d'un ensemble de termes ou de formules si tout unificateur σ_1 de cet ensemble est la composée de σ avec une autre substitution σ_2 ($\sigma_1 = sigma_2 \circ \sigma$).

On utilisera à plusieurs reprises le fait qu'une quelconque formule du langage du premier ordre puisse être représentée par un terme. L'opération qui permet ce passage consiste à transformer par renommage les symboles de prédicats et les quantificateurs et connecteurs logiques en symboles de fonctions. Elle est précisée par la définition ci-dessous.

Définition 101:

Soit $\mathcal{L} = (\mathcal{Q}, \mathcal{P}, \mathcal{F}, \mathcal{V}, \mathcal{C})$ et $\mathcal{L}' = (\mathcal{Q}', \mathcal{P}', \mathcal{F}', \mathcal{V}, \mathcal{C})$ deux langages du premier ordre construits tels que il existe une bijection θ de $\mathcal{Q} \cup \mathcal{P} \cup \mathcal{F}$ vers $\mathcal{F}'$ qui conserve l'arité des symboles. θ est prolongé sur les éléments de $\mathcal{V}$ et $\mathcal{C}$ par l'identité.

On note par un surlignement la fonction qui à une formule f de $\mathcal{L}$ associe le terme $\overline{f}$ de $\mathcal{L}'$ obtenu en appliquant θ à f.

Exemple 102:

Soit $\mathcal{L} = (Q = \{\vee, \ldots\}, P = \{P/2, Q/1\}, F = \{f/2, g/1\})$ et $\mathcal{L}' = (Q' = \varnothing, P = \varnothing, F = \{\vee'/2, p'/2, q'/1, f'/2, g'/1\})$. La bijection θ est définie par

$$\begin{array}{l} \vee/2 \rightarrow \vee'/2 \\ P/2 \rightarrow p'/2 \\ Q/1 \rightarrow q'/1 \\ f/2 \rightarrow f'/2 \\ g/1 \rightarrow g'/1 \end{array}$$

Sous ces conditions, $\overline{P(f(X,a), g(Y)) \vee Q(b)} = \vee'(p'(f'(X,a), g'(Y)), q'(b))$

On définit la notion de littéral le plus général pour désigner un littéral dont les arguments sont tous des variables.

Définition 103:

Un **littéral le plus général** est un littéral L tel que

$$\neg(\exists L', \exists \sigma,\ (\sigma(L') = L) \wedge (\nexists \sigma'\ L' = \sigma'(L))^{8})$$

On note $\boldsymbol{mgl(L)}$ le littéral le plus général correspondant à L.

Définition 104:

On appelle **datalog** l'ensemble des formules d'un langage du premier ordre sans symbole de fonction ($\mathcal{F} = \varnothing$) qui sont équivalentes à un ensemble de clause. On appelle **prolog** l'ensemble des formules d'un langage du premier ordre avec symboles de fonctions ($\mathcal{F} \neq \varnothing$) qui sont équivalentes à un ensemble de clauses.

Les termes datalog et prolog représentent donc des extensions des langages DATALOG et Prolog puisque ces derniers n'autorisent que des clauses de Horn. Nous utilisons ces termes sans majuscules pour bien les distinguer de ces langages. Nous insistons sur le fait que les formules auxquelles nous nous intéressons dans ces langages peuvent se mettre sous la forme de clauses quantifiées universellement.

[8] L' ne doit pas être un simple renommage des variables de L

9.2 Le principe de résolution au premier ordre

Le principe de résolution appliqué au premier ordre fait intervenir deux mécanismes. Le premier est la résolution proprement dite. Plus précisément appelé résolution binaire, il consiste à partir de deux clauses contenant des littéraux opposés et unifiables à en produire une troisième. Le second est un mécanisme de factorisation. À partir d'une clause contenant plusieurs occurrences d'un même littéral, il produit une clause ne contenant plus qu'une occurrence. Dans certaines présentations du principe de résolution, le mécanisme de factorisation est plus ou moins fortement incorporé au mécanisme de résolution. Pour notre part, nous séparerons clairement ces deux mécanismes car il ne posent pas les mêmes problèmes vis à vis de l'achèvement.

Cette séparation des deux mécanismes conduit à utiliser la notion de multi-ensemble pour la définition de ce qu'est une clause. En effet, nous ne souhaitons pas effectuer de factorisation implicite (fût-elle triviale). Il faut donc distinguer $\{P(X), P(X)\}$ de $\{P(X)\}$

Définition 105:

Une **clause** est un multi-ensemble de littéraux, et représente la disjonction de ces littéraux.

On s'autorise donc à répéter plusieurs fois un même littéral dans une clause. En revanche, on ne tient pas compte de l'ordre de ces littéraux.

Définition 106:

Soient deux clauses C_1 et C_2 qui ne partagent pas de variables et telles qu'il existe un littéral P vérifiant $P \in C_1$ et $\neg P \in C_2$

Si P et $\neg P$ ont un m.g.u. σ, la **résolution binaire** de C_1 et C_2 est la clause $\sigma((C_1 - \{P\}) \cup (C_2 - \{\neg P\}))$[9]

Exemple 107:

La résolution binaire de $A \vee P(X)$ et $A \vee \neg P(Y)$ est $A \vee A$ puisqu'on s'interdit en effet toute factorisation implicite ou explicite.

Celle de $P(X,Y) \vee Q(X,Y)$ et $\neg Q(Z,Z)$ donne $P(X,X)$.

Deux clauses qui partagent des variables peuvent être résolues à condition de renommer les variables auparavant.

Notation

On note $rename$ la fonction qui à une formule f associe une formule f' qui est une variante alphabétique de f et qui ne contient que des nouvelles variables (i.e. qui n'utilise pas de symbole de variable déjà employé par le passé)

Exemple 108:

$P(X,Y) \vee rename(Q(X,Z))$ représente $P(X,Y) \vee Q(T,U)$.

[9] L'opération de soustraction sur les multi-ensembles que nous utilisons ne retire qu'une occurrence à la fois. Par exemple, $\{A, A\} - \{A\} = \{A\}$.

Définition 109:

Soit C une clause contenant des occurrences P_1, P_2 d'un même littéral P. Si les occurrences P_1, P_2 ont un m.g.u. σ, la **factorisation** de la clause C est la clause $\sigma(C - \{P_2\})$.

Exemple 110:

La clause $P(a, X) \vee P(Y, b) \vee P(V, W)$ peut être factorisée en $P(a, X) \vee P(Y, b)$ ou $P(a, b) \vee P(V, W)$ ou encore $P(a, b)$.

La notion de fusion au premier ordre correspond à la notion de factorisation et est définie comme suit

Définition 111:

Un **littéral fusionné** l d'une clause C est un littéral

- tel que C est obtenue par une étape de factorisation à partir d'une clause C' et une substitution σ tels que $C = \sigma(C') - \{l\}$ (l est le littéral fusionné).
- ou tel qu'il existe $l_1 \in C, l_2 \in C$ (l_1 et l_2 apparaissant à des positions différentes de la clause) et une substitution σ tel que $\sigma(l_1) = \sigma(l_2) = l$ (une factorisation peut produire l)[10].

Exemple 112:

$P(a, b)$ est un littéral fusionné dans la clause $P(a, b) \vee Q$ obtenue par factorisation à partir de la clause $P(a, X) \vee P(Y, b) \vee Q$.

$P(a, b)$ est un littéral fusionné dans la clause $P(a, X) \vee P(Y, b)$. On remarquera que dans ce cas, ce littéral n'appartient pas à la clause.

On distingue au premier ordre différents types de subsomptions. La plus puissante est bien sûr l'implication.

Définition 113:

Une clause C_1 **implique** une clause C_2 si $C_1 \models C_2$. Une clause C_1 **subsume** une clause C_2 si il existe une substitution σ telle que $\sigma(C_1) = C_2$ Une clause C_1 **θ-subsume** une clause C_2 si il existe une substitution θ telle que $\theta(C_1) = C_2$ et $length(C_1) \leq length(C_2)$. Une clause C_1 **w-subsume** une clause C_2 si il existe une substitution σ telle que $\sigma(C_1) = C_2$ et $length(\sigma(C_1)) = length(C_1)$.

La w-subsomption est une restriction de la θ-subsomption qui interdit toute factorisation implicite. Il s'agit donc du critère de subsomption à employer pour simplifier une base achevée. La θ-subsomption est elle même une restriction de la subsomption. Enfin, cette dernière est aussi une restriction de l'implication.

Exemple 114:

La clause $\neg E(X) \vee E(s(X))$ implique la clause $\neg E(X) \vee E(s(s(X)))$. En revanche, il n'y a pas subsomption entre ces clauses.

La clause $P(a, X) \vee P(Y, b)$ subsume $P(a, b)$. En revanche, il n'y a pas θ-subsomption entre ces deux clauses.

[10]On définit ce deuxième cas de littéral fusionné pour pouvoir dire que «tout littéral conséquence d'une clause et d'un ensemble de faits et qui n'est pas fusionné peut être produit par chaînage avant». Sans cette deuxième partie de définition, $P(a, b)$ serait conséquence de $P(a, X) \vee P(Y, b)$ et non fusionné mais ne pourrait être produit par chaînage avant.

La clause $P(a, X) \vee P(Y, b)$ θ-subsume $P(a, b) \vee Q$. En revanche, il n'y a pas w-subsomption entre ces deux clauses.

La notion d'impliqué premier est similaire à celle du calcul propositionnel. Un impliqué premier est un impliqué qui n'est subsumé par aucun autre. Reste à définir la subsomption que l'on utilise. Si l'on choisit l'implication ou la subsomption, alors, il n'est pas toujours possible de trouver un impliqué premier qui subsume un impliqué donné [Mar93].

Exemple 115:

Soit la base $\mathcal{B}$ suivante

$$\begin{cases} ① \neg P(X, Y) \vee \neg P(Y, Z) \vee P(X, Z) \\ ② \neg P(X, X) \end{cases}$$

La clause $\neg P(X, X)$ est bien sûr un impliqué de $\mathcal{B}$. Cependant, aucun des impliqués qui la subsume n'est premier au sens de l'implication ou de la subsomption.

En effet, les clauses $C_n = \neg P(X_0, X_1) \vee \neg P(X_1, X_2) \vee \ldots \neg P(X_{n-1}, X_n) \vee \neg P(X_n, X_0)$ sont des impliqués de $\mathcal{B}$ (on peut le montrer en effectuant $n - 2$ résolutions avec ① puis une résolution avec ②). Chacune subsume (et donc implique) $\neg P(X, X)$ (il suffit de substituer chaque X_i par X). [Mar93] montre que ce sont les seuls impliqués de $\mathcal{B}$ qui impliquent $\neg P(X, X)$. Or, aucun C_n n'est un impliqué premier puisque chaque C_n est subsumé par C_{2n+1} (il suffit de substituer dans C_{2n+1} chaque $X_j, j > n$ par X_{j-n-1}).

Ce type de problème ne se pose pas quand on choisit la θ-subsomption ou sa restriction la w-subsomption. Comme la subsomption à considérer pour l'achèvement est la w-subsomption, nous adoptons la définition suivante.

Définition 116:

Un **impliqué premier** d'une base du premier ordre est un impliqué qui n'est w-subsumé par aucun autre.

Chapitre 10

Le chaînage avant en calcul des prédicats

Les algorithmes de chaînage avant en calcul des prédicats ne diffèrent guère de ceux en calcul propositionnel. Seule se superpose la notion de substitution. Ces algorithmes fonctionnent sur des bases de règles ou de clauses en forme prénexe quantifiée universellement. Les faits sont également quantifiés universellement. Aucun quantificateur existentiel ne doit apparaître.

10.1 Sur des règles

En calcul des prédicats, l'algorithme de chaînage avant sur des règles est le suivant.

1: **repeat**
2: **if** $\exists$ une règle $l_1, l_2, \ldots, l_n \rightarrow l$ et une substitution σ tels que chaque $\sigma(l_i)$ figure dans la base de faits **then**
3: **if** $\sigma(l)$ n'est pas θ-subsumé (au sens large) par un fait déjà présent **then**
4: ajouter $\sigma(l)$ aux faits
5: **end if**
6: **end if**
7: **until** une contradiction est détectée ou aucun fait n'a été ajouté

10.2 Sur des clauses

Comme en calcul propositionnel, le chaînage avant sur des clauses est très proche de l'*unit-resolution*.

1: **repeat**
2: **if** $\exists$ une clause $l_1 \vee l_2 \vee \ldots \vee l_n \vee l$ et une substitution σ tels que $\forall i$, $\sigma(\neg l_i)$ figure dans la base de faits **then**
3: **if** $\sigma(l)$ n'est pas θ-subsumé (au sens large) par un fait déjà présent **then**
4: ajouter $\sigma(l)$ aux faits

5: **end if**

6: **end if**

7: **until** une contradiction est détectée ou aucun fait n'a été ajouté

Chapitre 11

Le problème

Étendre la notion d'achèvement au premier ordre n'est pas tout à fait trivial. En effet, des problèmes de calculs infinis et d'indécidabilité surviennent alors qu'ils étaient absents du cas propositionnel. Ces problèmes nous obligent à raffiner la notion d'achèvement en particulier en ce qui concerne la complétude des inférences et l'équivalence entre la base initiale et la base achevée. Mais tout d'abord, il convient de vérifier que l'objectif de l'achèvement qui est de produire les littéraux impliqués par la connaissance est réalisable pour des bases du premier ordre.

11.1 L'ensemble des littéraux impliqués par une base est récursivement énumérable

Le but de l'achèvement au premier ordre est de permettre à un chaînage avant de calculer l'ensemble des faits impliqués par la connaissance. Encore faut-il que ce calcul soit possible pour une machine de Turing. Heureusement, il l'est pour le type de connaissance qui nous intéresse, à savoir des clauses en forme prénexe universellement quantifiées. Nous rappelons ci-dessous pourquoi.

Savoir si un littéral L (quantifié universellement) est impliqué par une base $\mathcal{B}$ de clauses en forme prénexe quantifiée universellement est décidable pour les bases datalog. Cela signifie qu'on sait dire en temps fini si oui ou non un littéral est impliqué par la base. En revanche, savoir si un littéral L (quantifié universellement) est impliqué par une base $\mathcal{B}$ de clauses en forme prénexe quantifiée universellement est semi-décidable pour les bases prolog. Cela signifie que si L est impliqué, on saura le montrer en un temps fini. En revanche, on ne sait pas toujours montrer que L n'est pas impliqué en un temps fini. Le point clef pour l'achèvement des bases prolog est que l'on sache prouver en temps fini (mais non borné à l'avance) qu'un littéral est impliqué. Ce résultat découle de la complétude réfutationnelle du principe de résolution au premier ordre [CL73]. En effet, un ensemble S de clauses est insatisfiable si et seulement si il existe une dérivation de la clause vide à partir de S. Cette dérivation est représentée par un arbre fini. Pour savoir si la clause vide peut être dérivée, il suffit de construire tous les arbres de résolution possibles en commençant par les arbres de plus faible hauteur. Cela peut se faire par exemple en procédant par saturation. Si la clause vide peut être produite, elle le sera après un temps fini. En revanche, rien ne nous permet dans le cas général de savoir si l'on peut cesser de rechercher un arbre produisant la clause vide, et dans ce cas, la procédure de réfutation boucle.

Comment alors produire l'ensemble des littéraux impliqués par une base si la procédure qui permet de tester l'implication d'un seul littéral peut ne pas terminer ? La solution consiste à simuler l'exécution en parallèle des tests d'implication, et de lancer une nouvelle exécution de test pour chacun des littéraux que l'on peut construire. On commence donc par énumérer les littéraux ce qui n'est guère difficile. Pour chaque littéral, on ajoute à une liste de machines de Turing en cours d'exécution la machine qui permet de tester l'implication de ce littéral. On simule alors un pas d'exécution de chacune des machines de la liste avant de passer au prochain littéral. Si l'une des machines de Turing prouve une implication, on ajoute le littéral à l'ensemble des impliqués et l'on peut alors retirer cette machine de la liste. À chaque instant, la liste des machines de Turing en cours d'exécution est finie. Cette simulation de parallélisme ne pose donc pas de problème puisqu'elle repose sur le même principe que le fonctionnement d'un système multitâches sur une machine monoprocesseur. Certes, la machine de Turing globale énumérant les littéraux impliqués sera de plus en plus lente, mais tout littéral impliqué sera produit en un temps fini. L'ensemble des littéraux impliqués par une base est donc bien récursivement énumérable. Ce résultat se généralise immédiatement au cas des clauses impliquées par la base, lui aussi récursivement énumérable.

Il faut bien noter que la formule $\neg P(X) \Rightarrow Q(X)$ signifie «si je peux prouver que $P(X)$ est faux, alors $Q(X)$ est vrai» et non «si je ne sais pas prouver que $P(X)$ est vrai, alors, $Q(X)$ est vrai». Cette dernière interprétation est la négation par l'échec et ne correspond pas aux définitions que nous utilisons. Du reste, cette interprétation permettrait de construire par programme le complémentaire d'ensembles récursivement énumérables. Comme ce complémentaire n'est pas toujours énumérable, il y aurait bien contradiction avec le fait que les faits impliqués sont récursivement énumérables.

11.2 Notion de complétude

La notion de complétude des inférences en calcul propositionnel était simple. Comme le chaînage avant s'exécutait en temps fini, être complet signifiait qu'une fois que l'algorithme s'était arrêté, l'ensemble des faits impliqués par la connaissance devait être contenu dans la base de faits. Or, le chaînage avant en calcul des prédicats peut ne jamais s'arrêter et produire une infinité de faits. Par exemple, sur la base $\{E(0), E(X) \rightarrow E(s(X))\}$ le chaînage avant produira une infinité de faits représentant l'ensemble des entiers. La notion de complétude des inférences doit donc changer.

Définition 117:

> Une implémentation du chaînage avant est dite **complète** pour une base de connaissances du premier ordre si pour tout littéral l impliqué par la connaissance, il existe un temps fini t_l à partir duquel ce littéral figure dans la base de faits ou est θ-subsumé par un littéral qui figure dans la base de faits.

Cette définition signifie que le chaînage avant sera considéré comme complet s'il sait énumérer l'ensemble des littéraux impliqués par la connaissance. Or, pour savoir énumérer tous les littéraux impliqués, l'implémentation du chaînage avant doit être équitable.

Définition 118:

> Une implémentation du chaînage avant est **équitable** si toute instance de règle ou de clause qui permet de prouver un littéral est effectivement utilisée au bout d'un temps fini pour prouver ce littéral.

On peut en effet imaginer des implémentations qui pour la base

$$\{E(0), E(X) \rightarrow E(s(X)), E(X) \rightarrow F(X)\}$$

commencent par appliquer systématiquement la deuxième règle pour générer l'infinité des entiers et seulement ensuite tentent d'utiliser la troisième règle. Il est évident que dans ce cas, aucun $F(X)$ ne serait produit en un temps fini.

On notera que la définition de complétude concerne essentiellement les bases de connaissances prolog. En effet, comme l'univers de Herbrand d'une base datalog est fini, l'ensemble des littéraux impliqués par cette connaissance est fini (à un renommage des variables près) et il suffit d'un temps fini pour les énumérer. On se retrouve donc presque dans la situation du calcul propositionnel.

Il est alors légitime de s'interroger sur l'utilité de l'achèvement pour les bases prolog. On peut en effet objecter à l'idée d'utiliser un chaînage avant qui peut boucler indéfiniment en produisant une infinité de conséquences. Il semblerait plus raisonnable dans ce cas d'utiliser une méthode guidée par le but comme le chaînage arrière. Cependant, même le chaînage arrière n'échappe pas aux problèmes de calculs infinis (qui n'a jamais écrit un programme Prolog qui boucle ?). Il est de plus tout à fait concevable de guider le chaînage avant pour aboutir plus rapidement à la production des faits qui nous intéressent (on peut par exemple établir des précédences entre les règles). On peut également stopper l'exécution du chaînage avant dès que les faits qui nous intéressent ont été produits. On le voit donc, la non finitude de l'ensemble des faits produits à partir d'une base prolog n'est pas un réel problème. De plus, lorsqu'une base est achevée, elle est complète pour l'unit propagation et donc également pour l'unit réfutation. L'achèvement permet donc également d'utiliser une base en chaînage arrière avec la même garantie de complétude qu'en chaînage avant.

Se limiter à l'achèvement des bases datalog semble d'autant plus inutile que prolog et datalog posent les mêmes problèmes de finitude pour l'achèvement, comme le montre la section suivante.

11.3 Finitude de la base achevée

La notion d'achèvement, on l'a vu, est basée sur le calcul d'un sous-ensemble des impliqués premiers. En calcul propositionnel, l'ensemble des impliqués premiers est toujours fini. Le premier problème qui s'oppose à l'extension de l'achèvement au calcul des prédicats est que l'ensemble des impliqués premiers d'une base du premier ordre peut être infini.

Exemple 119:

Soit la base suivante

- $E(X) \rightarrow E(s(X))$

  | *Si X est un entier, alors, le successeur de X ($s(X)$) est aussi un entier.*

- E(0).

  | *0 est un entier*

L'ensemble des impliqués premiers de cette base contient entre autres

$$E(s(0)),\ E(s(s(0))),\ E(s(s(s(0))))$$

et plus généralement toutes les clauses unitaires $E(s^n(0))$ pour toute valeur entière de n.

Le problème qui apparaît dans cet exemple est que l'usage de symboles de fonctions permet de construire une infinité de termes et par là même, une infinité d'impliqués premiers.

À première vue, on pourrait penser qu'en se limitant à des bases du premier ordre sans symbole de fonction (bases datalog) ce problème de finitude de l'ensemble des impliqués premiers serait évité. En fait, il n'en est rien.

Exemple 120:

Soit la base suivante

- $E(X) \wedge S(X, XX) \rightarrow E(XX)$

  | *Si X est un entier et que le successeur de X est XX, alors, XX est aussi un entier.*

L'ensemble des impliqués premiers de cette base contient les clauses suivantes :

$$\neg E(X_0) \vee \neg S(X_0, X_1) \vee E(X_1)$$
$$\neg E(X_0) \vee \neg S(X_0, X_1) \vee \neg S(X_1, X_2) \vee E(X_2)$$
$$\neg E(X_0) \vee \neg S(X_0, X_1) \vee \neg S(X_1, X_2) \vee S(X_2, X_3) \vee E(X_3)$$
$$\ldots$$
$$\neg E(X_0) \vee \neg S(X_0, X_1) \vee \ldots \vee \neg S(X_{n-1}, X_n) \vee E(X_n)$$
$$\ldots$$

Cet ensemble est infini, et aucune clause n'est θ-subsumée par une autre.

Démonstration 121:

Soit C_k la clause

$$\neg E(X_0) \vee \neg S(X_0, X_1) \vee \ldots \vee \neg S(X_{n-1}, X_n) \vee E(X_n)$$

Supposons que C_n θ-subsume C_m pour $n < m$. Soit σ la substitution qui remplace chaque variable X_i de C_m par un symbole de constante c_i nouveau à chaque fois. Si C_n θ-subsume C_m, il existe une substitution θ telle que $\theta(C_n) = C_m$ et donc, $\sigma \circ \theta(C_n) = \sigma(C_m)$.

On obtient immédiatement que $\sigma \circ \theta$ doit contenir la substitution $X_n \rightarrow c_m$ pour pouvoir unifier $E(X_n)$ et $E(c_m)$. En remontant alors la chaîne de relations successeur, on déduit que nécessairement, $\sigma \circ \theta$ doit contenir les substitutions $\{X_n \rightarrow c_m, X_{n-1} \rightarrow c_{m-1}, \ldots, X_0 \rightarrow c_{m-n}\}$. À ce moment, $\sigma \circ \theta(C_n)$ contient $\neg E(c_{m-n})$ tandis que $\sigma(C_m)$ contient $\neg E(c_0)$. Comme $m \neq n$, il n'existe pas de substitution θ telle que $\theta(C_n) = C_m$ et donc aucun C_n ne θ-subsume un autre C_m.

L'ensemble des impliqués premiers de cette base est infini et pourtant, aucun n'est utile puisque la base est achevée (il n'y a aucune fusion). Suffirait-il alors de trouver de bons critères d'élagages qui permettent d'extraire un sous-ensemble fini d'impliqués premiers nécessaires à l'achèvement ? Hélas non ! Il suffit de modifier très peu la base précédente pour obtenir un nombre infini d'impliqués premiers nécessaires à l'achèvement.

Exemple 122:

Soit la base suivante

- $N \rightarrow (E(X) \wedge S(X, XX) \rightarrow E(XX))$

> *Si tout est normal (i.e. si E code bien la propriété d'être un entier et S la relation successeur), alors, si X est un entier et que le successeur de X est XX, XX est aussi un entier.*

Les clauses conséquences de cette base sont

$$\neg N \vee \neg E(X_0) \vee \neg S(X_0, X_1) \vee E(X_1)$$
$$\neg N \vee \neg E(X_0) \vee \neg S(X_0, X_1) \vee \neg S(X_1, X_2) \vee E(X_2)$$
$$\neg N \vee \neg E(X_0) \vee \neg S(X_0, X_1) \vee \neg S(X_1, X_2) \vee \neg S(X_2, X_3) \vee E(X_3)$$
$$\ldots$$
$$\neg N \vee \neg E(X_0) \vee \neg S(X_0, X_1) \vee \ldots \vee \neg S(X_{n-1}, X_n) \vee E(X_n)$$
$$\ldots$$

Quel que soit n, il est clair que

$$\mathcal{B}, E(0), S(0,1), S(1,2), S(2,3), \ldots, S(n-1,n), \neg E(n) \models \neg N$$

Parmi toutes les clauses conséquences de $\mathcal{B}$, seule la clause C_n égale à $\neg N \vee \neg E(X_0) \vee \neg S(X_0, X_1) \vee \ldots \vee \neg S(X_{n-1}, X_n) \vee E(X_n)$ permet par l'une de ses variantes d'effectuer la déduction ci-dessus en chaînage avant.

Donc, quel que soit n, la base $\mathcal{B}$ n'est pas totalement achevée si elle ne contient pas la clause C_n.

De ce fait, $\mathcal{B}$ ne peut avoir de base achevée finie équivalente.

Il paraît donc impossible d'achever totalement et de manière finie certaines bases, même en se limitant à des cas datalog très simples. Cette impossibilité provient de ce que l'on sait également coder la notion de fonction en datalog, même en l'absence de symbole de fonction. Nous reviendrons sur ce point dans la section suivante.

Il existe alors plusieurs alternatives face à cette non finitude de la base achevée :

- se restreindre aux bases qui admettent un achèvement total équivalent fini
- ne pas imposer de restrictions sur les bases. Dans les cas où il n'existe pas de base équivalente, finie et achevée, on peut
  - soit effectuer un achèvement partiel, en limitant par exemple le nombre de faits pouvant être présents à l'exécution.
  - soit effectuer un achèvement total mais renoncer à l'équivalence entre $\mathcal{B}$ et $Achvt(\mathcal{B})$.

Il est a priori choquant de renoncer à l'équivalence de la base et de son achèvement puisque cela semble signifier renoncer soit à la complétude, soit à la correction des inférences, concessions qui sont toutes deux inacceptables dans l'optique de l'achèvement. En fait, cette toute dernière solution consiste à remplacer la contrainte d'équivalence sémantique entre la base et son achevé par une contrainte d'équivalence plus faible, comme nous le verrons prochainement.

11.4 Équivalence datalog-prolog

Du point de vue de l'achèvement, prolog et datalog posent les mêmes problèmes de finitude et de décidabilité. En effet, tout symbole de fonction f d'arité n peut être simulé par un prédicat F d'arité $n+1$. Les n premiers arguments de ce prédicat sont les paramètres de la fonction et le dernier est le résultat de la fonction. L'interprétation attendue de ce

prédicat $F(X_1, \ldots, X_n, X_{n+1})$ est d'être vrai si et seulement si $X_{n+1} = f(X_1, \ldots, X_n)$. La fonction est alors définie sous la forme d'une relation.

Toute clause (et donc toute règle) contenant des symboles de fonctions peut se transformer ainsi en une clause plus longue mais sans symbole de fonction, qui, sous réserve de l'ajout de certains faits, aura la même sémantique.

L'algorithme de transformation est celui-ci

1: **while** il existe un symbole de fonction dans la clause **do**
2: chercher $f(X_1, \ldots, X_n)$ la première occurrence d'un symbole de fonction dans la clause
3: choisir un nouveau symbole de variable Z qui n'apparaît pas dans la clause
4: remplacer l'occurrence de f par Z
5: ajouter à la clause le littéral $\neg F(X_1, \ldots, X_N, Z)$ tel que F soit le prédicat correspondant à f.
6: **end while**

Exemple 123:

La clause $P(f(X,Y)) \vee Q(g(f(Y,a)))$ sera transformée en $\neg F(X,Y,M) \vee \neg F(Y,a,N) \vee \neg G(N,O) \vee P(M) \vee Q(O)$

Soient C une clause avec symboles de fonctions et C' sa transformée par l'algorithme précédent. À toute interprétation qui satisfait C correspond une interprétation qui satisfait C' (il suffit de choisir comme interprétation des prédicats $F, G, \ldots$ une relation fonctionnelle). Réciproquement à toute interprétation de C' qui soit fonctionnelle pour les prédicats $F, G \ldots$ correspond une interprétation de C. Ces deux points permettent donc de définir intuitivement une forme d'équivalence sémantique entre la clause avec symboles de fonctions et sa transformée. Cette forme d'équivalence s'étend trivialement à un ensemble de clause. Ainsi est-il possible de coder en datalog tout problème codé en prolog.

Cela ne signifie pas pour autant que les deux langages ont le même pouvoir calculatoire. En effet, un programme datalog ne peut être considéré comme équivalent à un programme prolog que si toutes les relations codant les fonctions sont présentes. Cela suppose l'ajout d'un ensemble infini de faits, ce qui n'est pas permis en datalog.

Il faut cependant bien noter qu'ajouter une base de faits infinie à un programme datalog (par exemple, les relations de successions entre entiers) permet d'obtenir la puissance d'une machine de Turing et les problèmes d'indécidabilité qui y sont liés. Tout comme en prolog, l'implication devient semi-décidable si la base de faits est infinie. En effet, pour décider de l'implication, il suffit d'exhiber un sous-ensemble de la base qui soit fini et insatisfiable (théorème de compacité). Comme les sous-ensembles finis sont récursivement énumérables, il existe une procédure mécanique qui pourra démontrer l'implication après un temps fini. En revanche, pour infirmer l'implication, il faudrait montrer que tout sous-ensemble fini de la base est satisfiable. Cela ne peut se faire en temps fini, et donc, dans le cas général il est impossible d'infirmer une implication. Par contre, si la base de faits est finie (ainsi que le programme) énumérer les sous-ensembles de la base nécessite un temps fini et l'implication devient décidable.

Bien sûr, on ne souhaite pas faire fonctionner datalog avec une base de faits infinie. En revanche, on souhaite obtenir un achèvement qui soit valable pour toute base de faits finie. Cela signifie qu'on ne peut pas borner lors de la compilation la taille de la base de faits, et revient en fait à devoir travailler sur une base éventuellement infinie. Cela est analogue au fait qu'une union dénombrable d'ensembles fini peut donner un ensemble infini.

11.5 Extension du vocabulaire

On a vu que dans certains cas, il est nécessaire d'ajouter un nombre infini de clauses pour obtenir un achèvement total équivalent. Si l'on souhaite conserver un achèvement total et obtenir une base finie, il faut représenter l'ensemble infini de clauses à rajouter par un nombre fini de nouvelles clauses. Par exemple, il faudrait pouvoir représenter l'ensemble

$$\{S(0,X_1) \wedge \ldots \wedge S(X_{n-1},X_n) \wedge \neg E(X_n) \rightarrow \neg N | n \in \mathbb{N}\}$$

de manière finie. On pourrait par exemple utiliser une définition intentionnelle de la forme «les règles qui ont pour condition une chaîne de relations successeur commençant en 0 et se terminant en X et le prédicat $\neg E(X)$ et avec pour conclusion $\neg N$» Cette définition intentionnelle est bien sûr finie. On peut la transcrire au premier ordre par la définition récursive ci-dessous (le prédicat T(X) a pour signification «il existe une chaîne transitive de relations successeurs entre 0 et X»)

$$\begin{array}{l} S(0,X) \rightarrow T(X) \\ T(X) \wedge S(X,XX) \rightarrow T(XX) \\ T(X) \wedge \neg E(X) \rightarrow \neg N \end{array}$$

On peut donc représenter les inférences codées par un ensemble infini d'impliqués premier par un ensemble fini de clauses à condition d'augmenter le vocabulaire de la base. Cependant, cette modification du vocabulaire ne conserve pas l'équivalence sémantique habituelle. Il faut donc utiliser une nouvelle notion d'équivalence qui sera plus faible.

11.6 Notions d'équivalence

On peut définir plusieurs notions d'équivalence entre une base $\mathcal{B}$ et une base avec un vocabulaire étendu $\mathcal{B}'$. Chacune fait intervenir une notion de projection. En effet, pour pouvoir comparer des résultats obtenus à partir de $\mathcal{B}$ et $\mathcal{B}'$, il faut d'abord projeter ces derniers sur le vocabulaire de $\mathcal{B}$.

Dans chacune des notions d'équivalence présentées, on considère que le vocabulaire qui a servi à étendre $\mathcal{B}$ est disjoint du vocabulaire de toute base de faits.

La première notion d'équivalence concerne directement le chaînage avant et stipule que deux bases sont équivalentes si, pour tout jeu de faits, l'ensemble des littéraux de $\mathcal{B}$ dérivés par chaînage avant est le même sur les deux bases.

Définition 124:

On dit qu'une base $\mathcal{B}'$ est **équivalente par son comportement** à une base $\mathcal{B}$ (noté $\mathcal{B} \equiv_C \mathcal{B}'$, avec un C pour Comportement) si pour toute base de faits F, les littéraux impliqués par $\mathcal{B} \cup F$ sont les mêmes que la projection sur $Voc(\mathcal{B} \cup F)$ des littéraux impliqués par $\mathcal{B}' \cup F$. Si $\mathcal{B} \cup F$ est inconsistant, $\mathcal{B}' \cup F$ doit l'être aussi.

$$\mathcal{B} \equiv_C \mathcal{B}'$$
$$\Leftrightarrow$$
$$\forall F \text{ base de faits}, \begin{cases} \mathcal{B} \cup F \not\models \square \Rightarrow \{l | \mathcal{B} \cup F \models l\} = \{l | (\mathcal{B}' \cup F \models l) \wedge (Voc(l) \subseteq Voc(\mathcal{B} \cup F))\} \\ \mathcal{B} \cup F \models \square \Rightarrow \mathcal{B}' \cup F \models \square \end{cases}$$

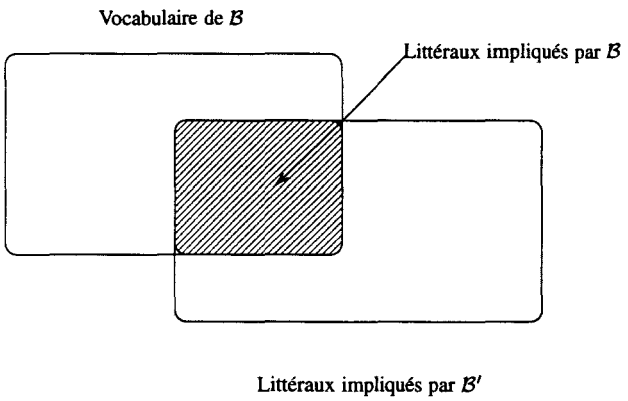


Figure 46: Projection des littéraux impliqués sur le vocabulaire de $\mathcal{B}$

On peut aussi définir une autre notion d'équivalence basée sur la notion d'impliqué. Deux bases sont équivalentes si la projection de leurs impliqués sur le vocabulaire de $\mathcal{B}$ sont les mêmes.

Définition 125:

La base $\mathcal{B}'$ est **équivalente par ses impliqués** à la base $\mathcal{B}$ (noté $\mathcal{B} \equiv_I \mathcal{B}'$, avec un I pour Impliqué) si et seulement si l'ensemble des impliqués de $\mathcal{B}'$ qui n'utilisent que le vocabulaire de $\mathcal{B}$ est égal à l'ensemble des impliqués de $\mathcal{B}$.

$$\mathcal{B} \equiv_I \mathcal{B}' \Leftrightarrow \{C | (\mathcal{B}' \models C) \wedge (Voc(C) \subseteq Voc(\mathcal{B}))\} = \{C | \mathcal{B} \models C\}$$

Enfin, une dernière notion d'équivalence est celle basée sur les impliquants. Deux bases sont équivalentes si la projection de leurs impliquants sur le vocabulaire de $\mathcal{B}$ sont les mêmes.

Définition 126:

La base $\mathcal{B}'$ est **équivalente par ses impliquants** à la base $\mathcal{B}$ (noté $\mathcal{B} \equiv_M \mathcal{B}'$ avec un M pour Modèles) si et seulement si la projection des impliquants de $\mathcal{B}'$ sur le vocabulaire de $\mathcal{B}$ est égale à l'ensemble des impliquants de $\mathcal{B}$.

$$\mathcal{B} \equiv_M \mathcal{B}' \Leftrightarrow \{M \cap Voc(\mathcal{B}) | M \models \mathcal{B}'\} = \{M | M \models \mathcal{B}\}$$

Ces trois notions d'équivalence sont en fait équivalentes.

Proposition 127:

$$\forall \mathcal{B}, \forall \mathcal{B}',\ \mathcal{B} \equiv_C \mathcal{B}' \Leftrightarrow \mathcal{B} \equiv_I \mathcal{B}' \Leftrightarrow \mathcal{B} \equiv_M \mathcal{B}'$$

Démonstration 128:

On montre d'abord que $\mathcal{B} \equiv_C \mathcal{B}' \Leftrightarrow \mathcal{B} \equiv_I \mathcal{B}'$.

1. $\mathcal{B} \equiv_I \mathcal{B}' \Rightarrow \mathcal{B} \equiv_C \mathcal{B}'$
   Soit F une base de faits

(a) $\mathcal{B} \cup F \models \square$
Il existe un impliqué premier P de $\mathcal{B}$ qui produit la clause vide par résolution avec les éléments de F. Par l'hypothèse $\mathcal{B} \equiv_I \mathcal{B}'$, P est aussi un impliqué premier de $\mathcal{B}'$. Donc $\mathcal{B}' \cup F \models \square$

(b) $\mathcal{B} \cup F \not\models \square$

i. $\{l | \mathcal{B} \cup F \models l\} \subseteq \{l | (\mathcal{B}' \cup F \models l) \wedge (Voc(l) \subseteq Voc(\mathcal{B} \cup F))\}$
Soit L tel que $Voc(L) \subseteq Voc(\mathcal{B} \cup F)$ et tel que $\mathcal{B} \cup F \models L$. Par la proposition 140, il existe un impliqué premier P de $\mathcal{B}$ tel que $P \cup F \models L$. Or, puisque $\mathcal{B} \equiv_I \mathcal{B}'$, P est aussi un impliqué premier de $\mathcal{B}'$. Donc $\mathcal{B}' \cup F \models L$.

ii. $\{l | \mathcal{B} \cup F \models l\} \supseteq \{l | (\mathcal{B}' \cup F \models l) \wedge (Voc(l) \subseteq Voc(\mathcal{B} \cup F))\}$
Soit L tel que $Voc(L) \subseteq Voc(\mathcal{B} \cup F)$ et $\mathcal{B}' \cup F \models L$. Par la proposition 140, il existe un impliqué premier P de $\mathcal{B}'$ tel que $P \cup F \models L$. Comme F et L n'utilisent pas le vocabulaire étendu de $\mathcal{B}$, on a $Voc(P) \subseteq Voc(\mathcal{B})$. Or, puisque $\mathcal{B} \equiv_I \mathcal{B}'$, P est aussi un impliqué premier de $\mathcal{B}$. Donc $\mathcal{B} \cup F \models L$.

Donc, $\mathcal{B} \equiv_I \mathcal{B}' \Rightarrow \mathcal{B} \equiv_C \mathcal{B}'$.

2. $\mathcal{B} \equiv_C \mathcal{B}' \Rightarrow \mathcal{B} \equiv_I \mathcal{B}'$

(a) $\{C | (C \in PI(\mathcal{B}')) \wedge (Voc(C) \subseteq Voc(\mathcal{B}))\} \subseteq PI(\mathcal{B})$
Soit $C \in PI(\mathcal{B}')$ et tel que $Voc(C) \subseteq Voc(\mathcal{B})$. Soit C' la clause obtenue à partir de C en substituant à chaque symbole de variable un nouveau symbole de constante, différent de toute constante de $\mathcal{B}$ ou $\mathcal{B}'$. Par construction, $\mathcal{B}' \cup \neg C' \models \square$. Comme $\mathcal{B} \equiv_C \mathcal{B}'$, on obtient que $\mathcal{B} \cup \neg C' \models \square$ et donc $\mathcal{B}$ a également C pour impliqué.

(b) $\{C | (C \in PI(\mathcal{B}')) \wedge (Voc(C) \subseteq Voc(\mathcal{B}))\} \supseteq PI(\mathcal{B})$
Soit $C \in PI(\mathcal{B})$. Soit C' la clause obtenue à partir de C en substituant à chaque symbole de variable un nouveau symbole de constante, différent de toute constante de $\mathcal{B}$ ou $\mathcal{B}'$. Par construction, $\mathcal{B} \cup \neg C' \models \square$. Comme $\mathcal{B} \equiv_C \mathcal{B}'$, on obtient que $\mathcal{B}' \cup \neg C' \models \square$ et donc $\mathcal{B}'$ a également C pour impliqué.

On montre maintenant que $\mathcal{B} \equiv_I \mathcal{B}' \Leftrightarrow \mathcal{B} \equiv_M \mathcal{B}'$.

1. $\mathcal{B} \equiv_I \mathcal{B}' \Rightarrow \mathcal{B} \equiv_M \mathcal{B}'$

(a) $\{M \cap Voc(\mathcal{B}) | M \models \mathcal{B}'\} \subseteq \{M | M \models \mathcal{B}\}$
Soit M un impliquant de $\mathcal{B}'$ et $M' = M \cap Voc(\mathcal{B})$ sa projection sur le vocabulaire de $\mathcal{B}$. Par définition, M' est aussi un impliquant des impliqués de $\mathcal{B}'$ qui n'utilisent que le vocabulaire de $\mathcal{B}$. De ce fait, M' est un impliquant de $\{C | (\mathcal{B}' \models C) \wedge (Voc(C) \subseteq Voc(\mathcal{B}))\}$. Comme $\mathcal{B} \equiv_I \mathcal{B}'$, M' est aussi un impliquant des impliqués de $\mathcal{B}$ et donc un impliquant de $\mathcal{B}$.

(b) $\{M \cap Voc(\mathcal{B}) | M \models \mathcal{B}'\} \supseteq \{M | M \models \mathcal{B}\}$
Soit M un impliquant de $\mathcal{B}$. Supposons que M ne puisse pas être prolongé en M' impliquant de $\mathcal{B}'$. Cela signifie que $\mathcal{B}' \cup M$ est inconsistant. Dans ce cas, il existe une réfutation linéaire de $\mathcal{B}' \cup M$ telle que les résolutions unitaires avec les littéraux de M figurent tout au bas de l'arbre (cf. proposition 138). La dernière clause centrale C de cette résolution avant les résolutions avec les éléments de M est un impliqué de $\mathcal{B}'$ qui ne contient que des littéraux de $\mathcal{B}$ puisque les résolutions avec M peuvent tous les effacer. Comme $\mathcal{B} \equiv_I \mathcal{B}'$, C est aussi un impliqué de $\mathcal{B}$. Or, M n'est pas un impliquant de C puisque

$C \cup M \models \square$. Il y a donc contradiction et M peut toujours se prolonger en M', impliquant de $\mathcal{B}'$.

Ce prolongement peut s'effectuer sans ajouter de littéraux de $\mathcal{B}$. Supposons en effet que ce ne soit pas le cas. Soit la décomposition de M' en $M \cup M_B \cup M_{B'}$ telle que $M_B \subseteq Voc(\mathcal{B})$ et $M_{B'} \subseteq Voc(\mathcal{B}')$. S'il faut absolument ajouter un littéral de $\mathcal{B}$ pour prolonger l'impliquant, c'est que $\forall M_{B'}$, $M \cup M_{B'} \not\models \mathcal{B}'$. On peut supposer que chaque $M_{B'}$ est maximal, c'est à dire qu'il est impossible d'ajouter un littéral du vocabulaire de $\mathcal{B}'$ sans aboutir à une formule insatisfiable. Pour chaque $M_{B'}$ maximal, il existe donc un modèle I de $M \cup M_{B'}$ qui ne soit pas un modèle de $\mathcal{B}'$. Il existe donc au moins une clause D de $\mathcal{B}'$ que I ne satisfait pas. On s'intéresse à la partie de D qui n'appartient pas au vocabulaire de $\mathcal{B}$. Deux cas sont possibles :

i. $D \cap \complement Voc(\mathcal{B}) = \varnothing$
Dans ce cas, D est un impliqué de $\mathcal{B}'$ qui ne fait appel qu'au vocabulaire de $\mathcal{B}$. Comme $\mathcal{B}' \equiv_I \mathcal{B}$, D est aussi un impliqué de B. Donc $M \models D$ ce qui contredit l'hypothèse.

ii. $D \cap \complement Voc(\mathcal{B}) \neq \varnothing$
On étudie deux cas :

A. $\exists L \in (D \cap \complement Voc(\mathcal{B}))$ tel que $\mathcal{B}' \cup L \not\models \square$
L peut alors être ajouté à l'un des $M_{B'}$ pour obtenir un impliquant de D. Cela signifie qu'il faut choisir un autre $M_{B'}$ et recommencer le raisonnement ou bien contredit l'hypothèse selon laquelle les $M_{B'}$ sont maximaux.

B. $\forall L \in (D \cap \complement Voc(\mathcal{B}))$, $\mathcal{B}' \cup L \models \square$
Soit L l'un de ces littéraux. Comme $\mathcal{B}' \cup L \models \square$ il existe L' unifiable avec L tel que $\mathcal{B}' \models \neg L'$. À ce moment, il est possible d'effectuer une résolution entre D et $\neg L'$ pour obtenir une clause D' qui n'est pas davantage satisfaite par l'interprétation I. On reproduit alors récursivement le raisonnement précédent jusqu'à aboutir à une clause (éventuellement vide) qui ne fait appel qu'au vocabulaire de $\mathcal{B}$. Le fait que cette clause ne soit pas satisfaite par I contredit les hypothèses.

2. $\mathcal{B} \equiv_M \mathcal{B}' \Rightarrow \mathcal{B} \equiv_I \mathcal{B}'$

(a) $\{C | (\mathcal{B}' \models C) \wedge (Voc(C) \subseteq Voc(\mathcal{B}))\} \subseteq \{C | \mathcal{B} \models C\}$
Soit C un impliqué de $\mathcal{B}'$ n'utilisant que le vocabulaire de $\mathcal{B}$. Puisque $\mathcal{B} \equiv_M \mathcal{B}'$, tout impliquant de $\mathcal{B}$ peut être prolongé en un impliquant de $\mathcal{B}'$. Comme C est satisfait par tous les impliquants de $\mathcal{B}'$ et ne contient que des littéraux de $\mathcal{B}$, on peut affirmer que tout impliquant de $\mathcal{B}$ satisfait C. Donc, $\mathcal{B} \models C$.

(b) $\{C | (\mathcal{B}' \models C) \wedge (Voc(C) \subseteq Voc(\mathcal{B}))\} \supseteq \{C | \mathcal{B} \models C\}$
Soit C un impliqué de $\mathcal{B}$. Comme $\mathcal{B} \equiv_M \mathcal{B}'$, tout impliquant de $\mathcal{B}'$ peut être projeté sur le vocabulaire de $\mathcal{B}$ pour obtenir un impliquant de $\mathcal{B}$. Puisque C ne contient que des littéraux de $\mathcal{B}$, tous les impliquants de $\mathcal{B}'$ satisfont C. Donc, $\mathcal{B}' \models C$.

Chapitre 12

Des théorèmes

12.1 Préliminaires

On montre d'abord que, dans une résolution linéaire, il est possible de permuter résolutions et factorisations et ce, sous des conditions très souples.

Proposition 129:

Dans une résolution linéaire, il est possible de permuter deux résolutions binaires successives à condition que la seconde résolution ait pour pivot un littéral de la première clause centrale.

Démonstration 130:

On montre qu'on peut réécrire une portion de résolution linéaire comme suit :

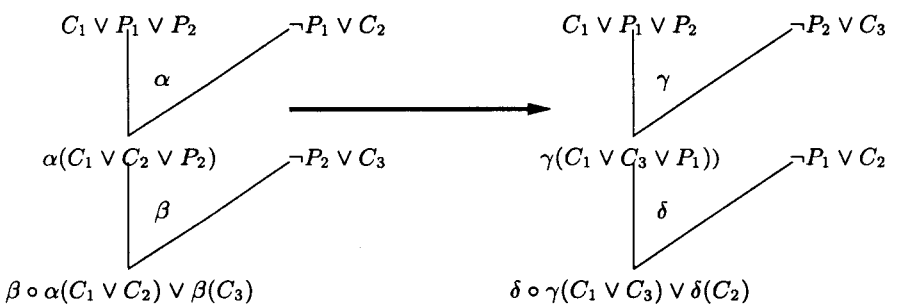


Figure 47: Permutation de deux résolutions binaires

Les unificateurs des pivots y ont été notés par des lettres grecques. De plus, aucune clause ne partage de variable avec une autre.

Par définition de l'unification, on a les équations suivantes

$$
\begin{aligned}
&|\alpha(P_1)| = |\alpha(\neg P_1)| \\
&|\beta \circ \alpha(P_2)| = |\beta(\neg P_2)| \\
&|\gamma(P_2)| = |\gamma(\neg P_2)| \\
&|\delta \circ \gamma(P_1)| = |\delta(\neg P_1)|
\end{aligned}
$$

Comme la clause $\neg P_2 \vee C_3$ a des variables différentes des clauses $C_1 \vee P_1 \vee P_2$ et $\neg P_1 \vee C_2$ (par construction de la résolution), l'application de la substitution α au prédicat $\neg P_2$ ne le modifie en rien. De ce fait, on peut écrire :

$$|\beta \circ \alpha(P_2)| = |\beta \circ \alpha(\neg P_2)|$$

Ceci signifie que $\beta \circ \alpha$ est un unificateur de $\{|P_2|, |\neg P_2|\}$. Or γ est un m.g.u. de cet ensemble. On en déduit donc que

$$\exists \epsilon, \beta \circ \alpha = \epsilon \circ \gamma$$

Si dans la résolution réécrite on choisit comme substitution appliquée à $\neg P_1 \vee C_2$ et $C_1 \vee C_2 \vee P_1$ la substitution ϵ au lieu du m.g.u. δ on obtient malgré tout un unificateur. En effet, puisque α est le m.g.u. de $|P_1|$ et $|\neg P_1|$, on a

$$|\epsilon \circ \gamma(P_1)| = |\beta \circ \alpha(P_1)| = |\beta(\alpha(\neg P_1))| = |\epsilon \circ \gamma(\neg P_1)|$$

La résolvante obtenue avec cet unificateur est $\epsilon \circ \gamma(C_1 \vee C_3) \vee \epsilon(C_2)$. Comme C_2 ne contient pas de variables sur lesquelles γ agisse, on peut écrire cette résolvante comme $\epsilon \circ \gamma(C_1 \vee C_2 \vee C_3)$ soit encore $\beta \circ \alpha(C_1 \vee C_2 \vee C_3)$.

Enfin, comme α ne porte pas sur les variables de C_2, on peut conclure

$$\beta \circ \alpha(C_1 \vee C_2) \vee \beta(C_3) = \beta \circ \alpha(C_1 \vee C_2 \vee C_3)$$

De ce fait, l'utilisation de ϵ comme unificateur permet d'obtenir la même résolvante. Le fait d'utiliser le m.g.u. δ au lieu de l'unificateur ϵ ne peut que produire une résolvante plus générale. Comme on peut de nouveau effectuer une permutation des deux résolutions binaires, on en déduit que la résolvante obtenue est exactement la même qu'avant la permutation.

Proposition 131:

Dans une résolution linéaire, il est possible de permuter une résolution binaire suivie d'une factorisation pour obtenir une factorisation suivie d'une résolution binaire, de sorte que la résolvante obtenue dans le second cas w-subsume celle obtenue dans le premier.

Démonstration 132:

On montre qu'il est possible d'effectuer la réécriture suivante et que la résolvante alors obtenue est plus générale. Dans cette figure, Q_1 et Q_2 sont deux instances d'un même prédicat Q.

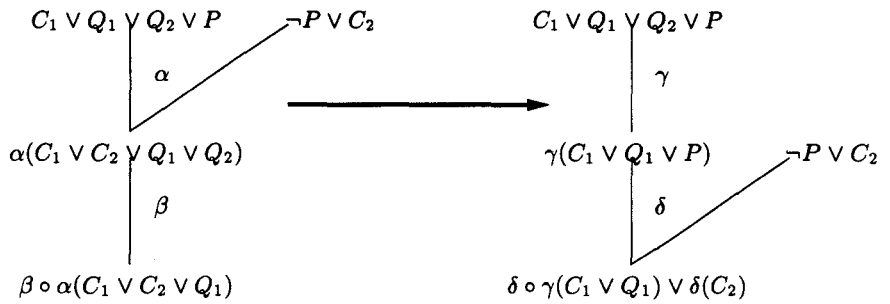


Figure 48: Descente d'une résolution binaire par permutation avec une factorisation

Les m.g.u. permettant les factorisations et résolutions sont notés par des lettres grecques.

Comme γ est le m.g.u. de $\{Q_1, Q_2\}$ et que $\beta \circ \alpha$ est un unificateur de ce même ensemble, il existe ϵ tel que $\epsilon \circ \gamma = \beta \circ \alpha$. De plus, ϵ est un unificateur de $|\gamma(P)|$ et $|\neg P|$. En effet, puisque γ n'agit pas sur les variables de P, et que α est un unificateur, on peut écrire

$$\epsilon(\neg P) = \epsilon \circ \gamma(\neg P) = \beta \circ \alpha(\neg P) = \beta \circ \alpha(P) = \epsilon \circ \gamma(P)$$

On peut donc bien unifier $|\gamma(P)|$ et $|\neg P|$ par ϵ et effectuer la résolution. Comme δ est un unificateur éventuellement plus général que ϵ, on peut conclure que la seconde résolvante w-subsume au sens large la première.

Proposition 133:

Dans une résolution linéaire, il est possible de permuter une factorisation suivie d'une résolution binaire pour obtenir une résolution binaire suivie d'une factorisation, de sorte que la résolvante obtenue dans le second cas w-subsume celle obtenue dans le premier, à la condition toutefois que la résolution ne porte pas sur le littéral factorisé.

Si cette dernière condition n'est pas vérifiée, et à condition que la résolution soit unitaire, on peut remplacer la factorisation suivie de la résolution unitaire par une succession de deux résolutions unitaires produisant la même résolvante.

Démonstration 134:

Dans le cas où la résolution binaire ne porte pas sur le littéral fusionné, on peut effectuer la réécriture suivante sans perdre en généralité.

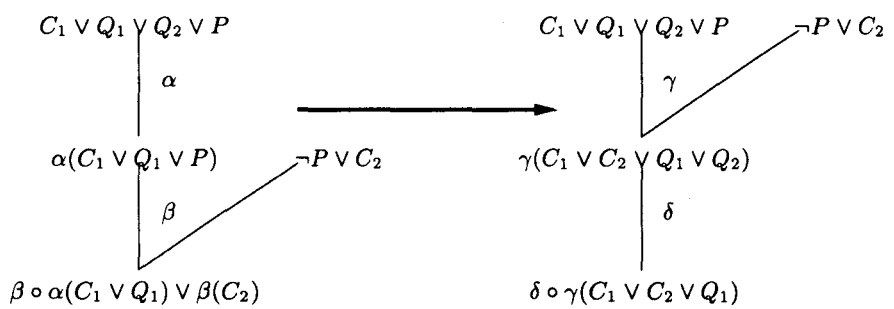


Figure 49: Montée d'une résolution binaire par permutation avec une factorisation (1)

En effet, $\beta \circ \alpha$ est un unificateur de $\{|P|, |\neg P|\}$. Puisque γ est un m.g.u. de ce même ensemble, il existe ϵ tel que $\epsilon \circ \gamma = \beta \circ \alpha$. La substitution ϵ unifie $\{\gamma(Q_1), \gamma(Q_2)\}$ puisque

$$\epsilon \circ \gamma(Q_1) = \beta \circ \alpha(Q_1) = \beta \circ \alpha(Q_2) = \epsilon \circ \gamma(Q_2)$$

Donc, la factorisation de $\gamma(C_1 \vee Q_1 \vee Q_2)\}$ est possible et donne $\beta \circ \alpha(C_1 \vee Q_1)$. Comme δ est plus générale que ϵ, la résolvante dans le cas réécrit w-subsume au sens large l'autre résolvante.

Si la résolution porte sur le littéral fusionné et est unitaire, on effectue la réécriture suivante

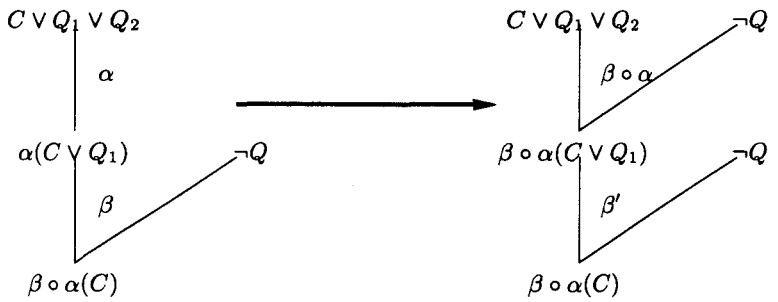


Figure 50: Montée d'une résolution binaire par permutation avec une factorisation (2)

La substitution β' est obtenue en restreignant β aux variables de $\neg Q$ après composition d'une substitution qui permet d'annuler le renommage des variables dans la seconde occurrence de la clause $\neg Q$.

On retrouve ce dernier résultat sous une forme plus générale dans [Nol80, p 252].

Note 135:

Lorsque la résolution n'est pas unitaire, il n'est pas toujours possible de descendre la factorisation comme le montre cet exemple

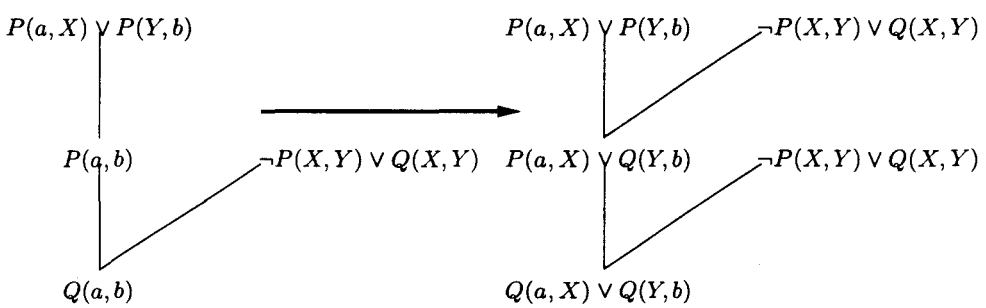


Figure 51: Exemple de factorisation qui ne peut être descendue

12.2 L'ajout des impliqués premiers est une méthode d'achèvement

On doit d'abord montrer que l'ajout de l'ensemble des impliqués premiers d'une base est une méthode d'achèvement, c'est à dire que :

$$\forall \mathcal{B}, \forall F, \forall L, (\mathcal{B} \cup F \models L) \Rightarrow (\exists C \in PI(\mathcal{B}), \exists L' \in Fwch(C \cup F), \exists \sigma,\ \sigma(L') = L)$$

On montre tout d'abord que toute clause impliquée est produite par résolution linéaire.

Proposition 136:

Si $\mathcal{B} \models C$, et C n'est pas une tautologie, il existe une résolution linéaire à partir de $\mathcal{B}$ qui produit une résolvante w-subsumant C.

Démonstration 137:

Découle de [MR72] ou [Ino92, théorème 4.3, p 94]

Ce résultat est en particulier vrai si la clause C est unitaire. On cherche alors à transformer une résolution linéaire en une résolution se terminant par une succession de résolutions unitaires afin de se rapprocher du fonctionnement du chaînage avant.

Proposition 138:

Si $\mathcal{B} \cup F \models C$, il existe une résolution linéaire qui produit une résolvante qui w-subsume C et qui est de la forme $\{B_1, B_2, \ldots, B_n, F_1, F_2, \ldots F_m\}$ où $\forall i, B_i \in \mathcal{B}$ et $\forall i, F_i \in F$.

Démonstration 139:

Puisque $\mathcal{B} \cup F \models C$ et par la proposition 136, il existe une résolution linéaire à partir d'un sous-ensemble de clauses $\{B_1, B_2, \ldots, B_n\}$ de $\mathcal{B}$ et d'un sous ensemble $\{F_1, F_2, \ldots F_m\}$ de F qui produit une clause w-subsumant C. Par la proposition 129 et 131, les résolutions utilisant les faits peuvent être reportées en fin de résolution pour fournir une résolvante au moins aussi générale.

On en déduit le résultat attendu :

Proposition 140:

L'ajout des impliqués w-premiers d'une base au premier ordre est une méthode d'achèvement.

Démonstration 141:

Pour toute base $\mathcal{B}$, tout ensemble de faits F et tout littéral P pour lesquels $\mathcal{B} \cup F \models L$, il découle de la proposition 138 qu'il existe une résolution linéaire à partir des clauses $\{B_1, B_2, \ldots, B_n\}$ de $\mathcal{B}$ qui produit un impliqué C, et, à partir de C, une succession de résolutions unitaires avec les faits $\{F_1, F_2, \ldots F_m\}$ de F qui produit un littéral plus général que L.

De ce fait, l'ensemble des impliqués d'une base est un achèvement de celle-ci.

Si C est subsumée par une clause C', il existe une substitution σ telle que $\sigma(C') \subseteq C$. On peut alors construire une résolution linéaire à partir de C' et un sous-ensemble de $\{F_1, F_2, \ldots F_m\}$ qui produit L ou la clause vide. En effet, si σ ne permet pas d'effectuer de factorisation, il suffit de reporter l'effet de σ sur chacun des unificateurs utilisés dans les résolutions unitaires, et on obtient alors exactement L. Si σ permet une factorisation, on construit une résolution linéaire qui commence par une factorisation et se poursuit

par des résolutions unitaires. Par la proposition 133, la factorisation peut être éliminée. La résolvante obtenue subsume (au sens large) alors L, puisque C' subsume C.

Donc, l'ensemble des impliqués premiers d'une base forme un achèvement de celle-ci.

12.3 Condition nécessaire et suffisante d'achèvement

On montre qu'un littéral non fusionné impliqué par une clause peut être produit par chaînage avant.

Proposition 142:

Soit une clause C, un ensemble de faits F et un littéral l tels que $C \cup F \models l$.

Si l n'est pas un littéral fusionné dans C, alors, $\exists l', \exists \sigma$, tel que $(l' \in Fwch(C \cup F)) \wedge (\sigma(l') = l)$

Démonstration 143:

Puisque $C \cup F \models l$ et par la proposition 136, il existe une résolution linéaire à partir de $C \cup F$ qui produit un littéral l' subsumant l. Par la proposition 138, on peut obtenir une résolution ayant C pour racine. De plus, par la proposition 133, on peut supprimer toute étape de factorisation entre les résolutions avec les faits. Enfin, comme l n'est pas un littéral fusionné, on déduit que la résolution ne contient aucune étape de factorisation. De ce fait, le chaînage avant peut produire l'.

La réciproque de la proposition, à savoir l produit par chaînage avant implique l non fusionné dans C est fausse. Par exemple, si $C = P(a, X) \vee P(Y, b)$ et $F = \{\neg P(c, b)\}$ et bien que $P(a, b)$ soit un littéral fusionné dans C, il existe un littéral produit par chaînage avant $(P(a, X))$ qui subsume ce littéral fusionné.

Proposition 144:

Soient C_1 et C_2 deux clauses ainsi que R leur résolvante binaire sur le pivot P. Soient F un ensemble de faits et l un littéral tels que $R \cup F \models l$.

Si l n'est pas un littéral fusionné dans R, alors, $\exists l', \exists \sigma$, tq $(l' \in Fwch(\{C_1, C_2\} \cup F)) \wedge (\sigma(l') = l)$

Démonstration 145:

On peut construire une résolution linéaire produisant l' à partir de R et F comme dans la proposition 142. On chapeaute alors cette résolution par l'étape de résolution entre C_1 et C_2 et l'on fait usage de la proposition 133. La résolution ainsi obtenue ne contient pas de factorisation. l' peut donc être produit par chaînage avant.

Proposition 146:

Soit L un littéral et R une résolvante obtenue par une résolution linéaire à partir d'un ensemble $\mathcal{B}$ de clauses et qui ne contient pas de factorisation produisant un ascendant de L ou produisant un littéral servant de pivot dans la résolution.

$$\forall F, \ (L \in Fwch(R \cup F)) \Leftrightarrow (L \in Fwch(\mathcal{B} \cup F))$$

Démonstration 147:

Par hypothèse, il existe une résolution linéaire à partir de R et F qui produit L. On peut lui greffer la résolution linéaire qui produit R à partir de $\mathcal{B}$ (transformation 1 de la figure ci-dessous). On peut alors transformer cette résolution linéaire en une succession de résolutions linéaires inputs (transformation 2). Par usage des propositions 129 et 133, on peut faire remonter les résolutions unitaires aussi près que possible de la première occurrence du pivot (transformation 3). Lors de cette étape, toute factorisation portant sur un littéral qui est effacé par la suite par résolution unitaire est remplacée par deux résolutions unitaires successives 133.

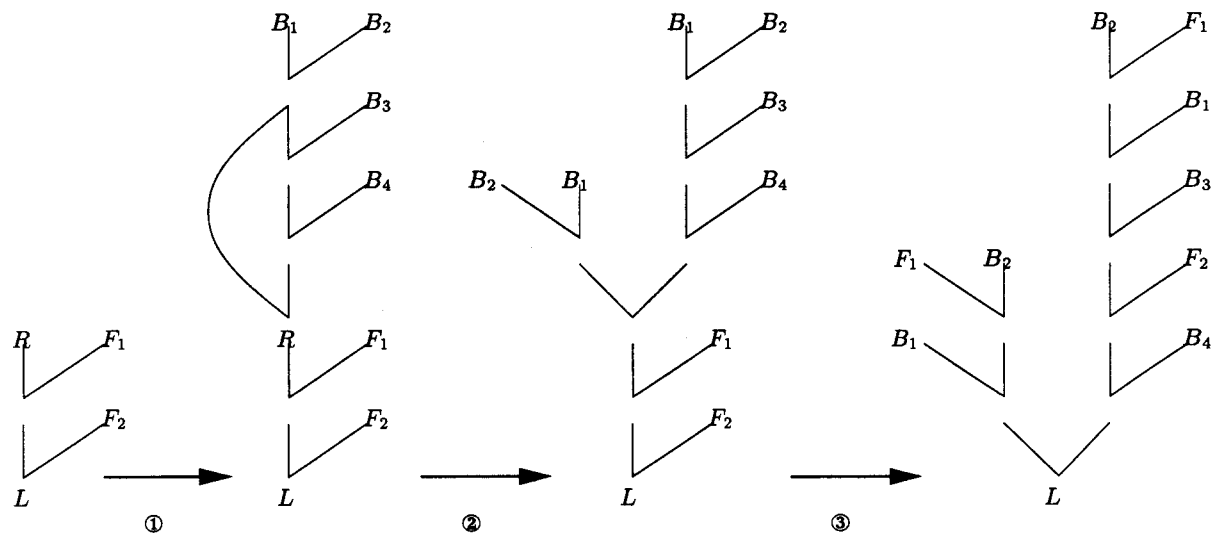


Figure 52: Transformations successives de l'arbre

La figure suivante présente un exemple propositionnel illustrant la succession de transformations effectuées.

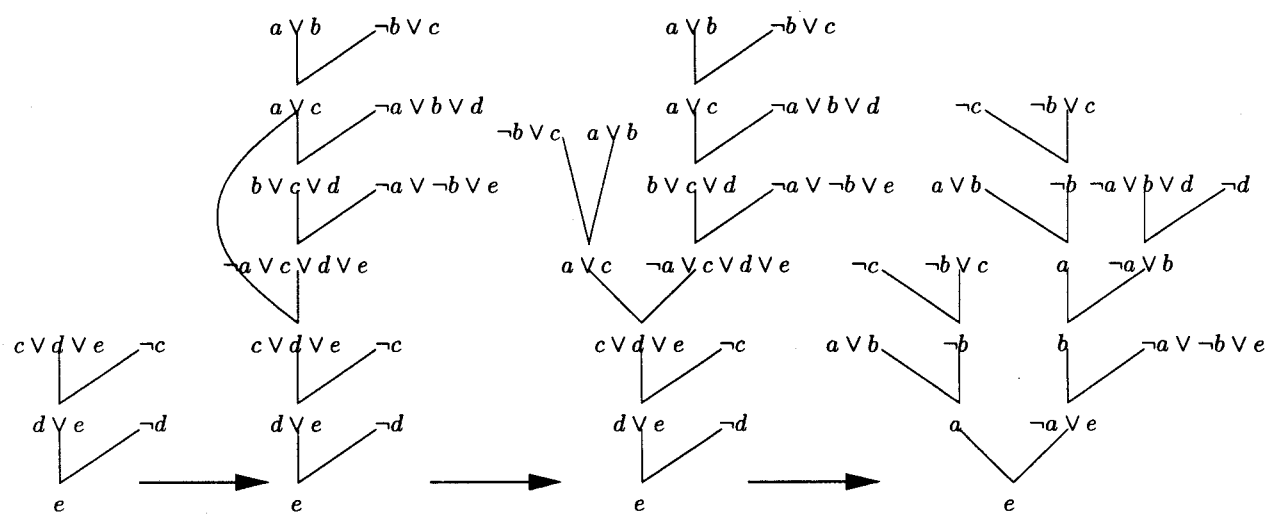


Figure 53: Un exemple concret de transformation

Sous les conditions énoncées dans la proposition, l'arbre obtenu représente une succession de résolutions unitaires à partir de $\mathcal{B} \cup F$ produisant L. En effet, les feuilles de cet

arbre sont soit des clauses de $\mathcal{B}$ soit des faits de F. De plus, cet arbre est constitué par l'assemblage de sous arbres de la forme

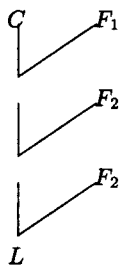


Figure 54: Constitution des sous-arbres

C représente une clause quelconque (i.e. unaire ou non), et les F_i et L représentent des clauses unitaires.

Nous prouvons cela par l'absurde. Supposons qu'il existe un sous-arbre qui ne puisse pas être prolongé pour aboutir à cette forme.

Ce sous-arbre se termine donc par une clause C' qui n'est pas unitaire. Comme la racine de l'arbre global est une clause unitaire, cette clause C' est suivie

- soit d'une opération de factorisation
  Soit F le littéral factorisé.
  On peut envisager les cas suivants
  - F ne sert jamais de pivot par la suite
    Dans ce cas, il n'est jamais effacé des résolvantes successives, et on en déduit que L est une instance de F. Ceci contredit l'hypothèse que L ne descend pas d'un littéral fusionné.
  - F sert de pivot par la suite
    - lors d'une résolution unitaire
      Cela signifie que les résolutions unitaires n'ont pas été remontées aussi haut que possible.
    - lors d'une résolution non unitaire
      Cela contredit l'hypothèse selon laquelle un littéral servant de pivot ne peut descendre d'un littéral factorisé.
- soit d'une résolution binaire
  Soit C'' la clause avec laquelle s'effectue la résolution et R leur résolvante.
  - C'' est unitaire
    Dans ce cas, le sous-arbre peut être prolongé ce qui contredit l'hypothèse.
  - C'' n'est pas unitaire
    La résolvante R ne peut être unitaire. Elle contient donc au moins deux littéraux dont l'un au moins E doit être éliminé par résolution ou factorisation (sans quoi la racine de l'arbre global ne serait pas une clause unitaire). E ne peut être éliminé par factorisation pour les raisons déjà indiquées. E ne peut non plus être éliminé par résolution unitaire car dans ce cas, cette résolution pourrait être remontée dans l'arbre. Enfin, une résolution binaire ne peut éliminer E sans introduire un autre littéral qu'il faudrait éliminer à son tour. On en déduit la contradiction.

Chapitre 13

L'achèvement par méta-interprète

Une fois acceptée l'extension du vocabulaire pour permettre l'achèvement de la base, survient naturellement la possibilité d'ajouter un méta-interprète complet à la base pour permettre d'effectuer toutes les inférences possibles. Cette technique répond à la définition de l'achèvement. Elle ne présente cependant aucun intérêt pratique puisqu'il n'y a aucun espoir que les inférences sur la base achevée soient plus efficaces qu'avec une version non-méta de l'interprète complet écrite dans un langage efficace. Elle présente cependant l'intérêt de fixer une autre borne pour les procédures d'achèvement. En effet, si l'achèvement sans extension du vocabulaire n'est pas toujours possible, étendre trop fortement le vocabulaire conduit à un achèvement qui n'a plus d'intérêt pratique. Les algorithmes d'achèvement devront donc, tout comme dans le mythe d'Icare choisir la juste altitude entre les flots tumultueux et le soleil brûlant.

13.1 Le cas prolog

Nous étudions ci-dessous comment on peut réaliser un tel méta-interprète dans le cas prolog.

L'idée est de transcrire la base à achever en termes, qui pourront alors être traités par le méta interprète. C'est la phase de **traduction**. Le méta interprète calcule alors l'ensemble des littéraux impliqués sous forme de termes. Des clauses supplémentaires permettent de retransformer ces termes en littéraux de la base initiale. C'est la phase de **restitution**. Le schéma suivant illustre les trois étapes de ce calcul.

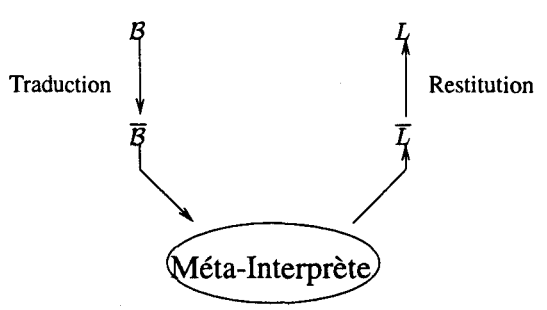


Figure 55: Phase de traduction et restitution en amont et aval du méta-interprète.

Nous détaillons maintenant les trois phases.

1. La traduction
   Il faut d'abord traduire l'ensemble des clauses de la base en termes manipulables par le méta-interprète. On transforme donc chaque clause

   $$P(X_1,\ldots,X_n) \vee Q(Y_1,\ldots,Y_m)\ldots$$

   de la base $\mathcal{B}$ en une clause unitaire

   $$\texttt{clause(}[\overline{P}(X_1,\ldots,X_n),\overline{Q}(Y_1,\ldots,Y_m),\ldots]\texttt{)}$$

   La clause de la base initiale est ainsi codée par la liste de ses littéraux. Les variables, constantes et symboles de fonctions de la clause initiale ne sont pas modifiés par la transformation. Seuls les symboles de prédicats P sont transformés en $\overline{P}$. Cette transformation s'effectue une fois pour toutes à la compilation.
   Il faut de plus pouvoir traduire les faits en termes du niveau méta. Cela doit s'effectuer à l'exécution. On ajoute donc les règles suivantes pour chaque prédicat P d'arité n :

   $\texttt{P}(X_1,\ldots,X_n)\rightarrow\texttt{clause}([\overline{P}(X_1,\ldots,X_n)])$.
   $\neg\texttt{P}(X_1,\ldots,X_n)\rightarrow\texttt{clause}([\texttt{not}(\overline{P}(X_1,\ldots,X_n))])$.

2. Le méta-interprète.
   Nous utilisons ici un méta-interprète des plus simples et inefficaces puisqu'il effectuera une simple saturation par résolution. Cependant, comme la résolution est complète pour le calcul des impliqués premiers, il satisfait pleinement notre but.
   Nous présentons d'abord des prédicats d'usage général.
   Le prédicat $\texttt{diff}(X,Y)$ est vrai lorsque X et Y représentent des entiers différents codés en unaire.

   $\texttt{diff}(s(X),0)$.
   $\texttt{diff}(0,s(X))$.
   $\texttt{diff}(X,Y)\rightarrow\ \texttt{diff}(s(X),s(Y))$.

   Le prédicat $\texttt{member}(X,L,P)$ réussit si X apparaît dans la liste L à la position numéro P.

   $\texttt{member}(X,[X|L],0)$.
   $\texttt{member}(X,L,P)\rightarrow\ \texttt{member}(X,[\_|L],s(P))$.

   La concaténation de deux listes s'effectue de manière classique :

   $\texttt{concat}(L,[],L)$.
   $\texttt{concat}(L_1,L_2,L_3)\rightarrow\ \texttt{concat}(L_1,[X|L_2],[X|L_3])$.

   On définit maintenant un prédicat $\texttt{remove}(P,L_1,L_2)$ qui réussit si L_2 est la liste L_1 privé de l'élément à la position P.

   $\texttt{remove(0,[X],[])}$.
   $\texttt{remove(0,[X|L],L)}$.
   $\texttt{remove}(s(P),L_1,L_2)\rightarrow\ \texttt{remove}(P,[Y|L_1],[Y|L_2])$.

   Les règles codant la résolution sont au nombre de trois :
   On définit le prédicat $\texttt{pivot}(L_1,L_2)$ qui teste si les deux littéraux L_1 et $\neg L_2$ peuvent s'unifier et, le cas échéant, effectue cette unification.

   $\texttt{pivot}(L,\texttt{not}(L))$.
   $\texttt{pivot}(\texttt{not}(L),L)$.

   La règle suivante encode la résolution binaire

   $\texttt{clause}(C_1)\wedge\texttt{clause}(C_2)\wedge\texttt{member}(L_1,C_1,P_1)\wedge\texttt{member}(L_2,C_2,P_2)$
   $\wedge\texttt{pivot}(L_1,L_2)\wedge\texttt{remove}(P_1,C_1,R_1)\wedge\texttt{remove}(P_2,C_2,R_2)$
   $\wedge\texttt{concat}(R_1,R_2,R)\rightarrow\ \texttt{clause(R)}$.

On notera que le renommage des variables des clauses parentes de la résolvante s'effectue automatiquement grâce au mécanisme d'exécution du chaînage avant ou arrière. Par exemple, en Prolog, comme les clauses sont stockées dans la database et dans des faits distincts, il n'y a aucune liaison entre leurs variables. De ce fait, les clauses ont bien des variables disjointes.

La règle suivante code la factorisation

$\texttt{clause}(C) \wedge \texttt{member}(L, C, P_1) \wedge \texttt{member}(L, C, P_2)$
$\wedge \texttt{diff}(P_1, P_2) \wedge \texttt{remove}(P_2, C, F) \rightarrow \texttt{clause}(F)$.

3. La restitution.
   La restitution s'effectue très simplement en ajoutant, pour chaque prédicat P d'arité n, les règles suivantes :
   $\texttt{clause}([\overline{P}(X_1, \ldots, X_n)]) \rightarrow \texttt{P}(X_1, \ldots, X_n)$.
   $\texttt{clause}([\texttt{not}(\overline{P}(X_1, \ldots, X_n))]) \rightarrow \neg\texttt{P}(X_1, \ldots, X_n)$.

La base obtenue a bien un comportement équivalent à la base initiale. L'achèvement réalisé par ce méta-interprète est total. Il est de plus polynomial par rapport à la taille de la base. Cependant, l'exécution de cet interprète est très inefficace. Il s'agit donc là seulement d'un outil théorique.

13.2 Le cas datalog

Il est possible d'imiter la démarche précédente en datalog. Cependant, l'absence de symbole de fonction rend le problème beaucoup plus délicat puisqu'il n'est pas possible de gérer une liste quelconque de littéraux. En revanche, il est possible de gérer une liste de littéraux de longueur finie et bornée à l'avance.

Coder une clause de longueur au plus égale à n est facile en datalog. Il suffit de coder le nom des littéraux présents dans la clause dans le symbole de prédicat et de spécifier comme liste d'arguments celle des littéraux de la clause. Par exemple, pour coder la clause $P(X_1, \ldots, X_l) \vee Q(Y_1, \ldots Y_m) \vee S(Z_1, \ldots, Z_n)$ on écrira

$$clause_{P,Q,R}(X_1, \ldots, X_l, Y_1, \ldots Y_m, Z_1, \ldots, Z_n)$$

On essaye donc de travailler sur des clauses de tailles finies tout au long de l'exécution du méta-interprète. Cela revient à dire qu'on s'intéresse à des résolutions linéaires où la taille des clauses centrales et latérales est toujours bornée par n. Compte tenu de notre objectif qui est de produire l'ensemble des littéraux impliqués (ce sont des clauses de longueur 1) et de la possibilité d'effectuer les résolutions unitaires avec les faits au plus tôt, cette restriction n'apparaît pas à première vue tout à fait déraisonnable. Il n'est cependant pas évident que cette restriction soit complète.

À ce jour, on sait que n doit dépendre de la base que l'on traite et peut être arbitrairement grand. Considérons par exemple la forme normale conjonctive de $l \vee (l_1 \wedge \neg l_1) \vee (l_2 \wedge \neg l_2) \vee \ldots \vee (l_n \wedge \neg l_n)$. Pour $n = 3$, cette base s'écrit

$$\begin{cases} l \vee l_1 \vee l_2 \vee l_3 \\ l \vee l_1 \vee l_2 \vee \neg l_3 \\ l \vee l_1 \vee \neg l_2 \vee l_3 \\ l \vee l_1 \vee \neg l_2 \vee \neg l_3 \\ l \vee \neg l_1 \vee l_2 \vee l_3 \\ l \vee \neg l_1 \vee l_2 \vee \neg l_3 \\ l \vee \neg l_1 \vee \neg l_2 \vee l_3 \\ l \vee \neg l_1 \vee \neg l_2 \vee \neg l_3 \end{cases}$$

Il est clair sur la forme disjonctive de cette base que l est impliqué. Or, pour le produire par résolution, il faut obligatoirement produire une clause de la forme $l_1 \vee \ldots \vee l_n \vee l_1 \vee \ldots \vee l_n$ et ensuite effectuer les factorisations pour éliminer les littéraux doubles de cette clause. De ce fait, il faut au moins savoir gérer une clause de longueur $2n$ pour être complet sur ce type de base.

On sait également que, pour de très larges classes de bases, limiter la taille des clauses à n n'interdit pas d'être complet pour la production des littéraux impliqués.

Il reste donc à savoir

- s'il existe des bases pour lesquelles limiter la taille des clauses à n (fixé en fonction de la base) ne permet pas de produire tous les littéraux impliqués pour toutes les bases de faits possibles
- dans le cas où la réponse à la question précédente est négative, savoir s'il est possible de calculer un n convenable par algorithme

Cependant, l'intérêt de ces questions doit être nuancé par le peu d'intérêt pratique que présente une méthode d'achèvement datalog par méta-interprète. En effet, le nombre de clauses à ajouter pour obtenir un achèvement est beaucoup plus important que dans le cas prolog, comme on peut maintenant le constater.

Les phases du calcul sont les mêmes que dans le cas prolog :

1. Traduction
   Il s'agit de traduire chaque clause de la base en un prédicat datalog. Une clause $P_1(X_1, \ldots, X_{n_1}) \vee P_2(X_{n_1+1}, \ldots, X_{n_2}) \vee \ldots \vee P_n(X_{n_{n-1}+1}, \ldots, X_{n_n})$ se traduit en

   $$clause_{P_1,P_2,\ldots,P_n}(X_1, \ldots, X_{n_n})$$

   Il reste à traduire les faits à l'usage du méta interprète. Pour tout prédicat $P(X_1, \ldots X_n)$ de la base, on ajoute donc

   $$\begin{cases} P(X_1, \ldots, X_n) \rightarrow clause_P(X_1, \ldots, X_n) \\ \neg P(X_1, \ldots, X_n) \rightarrow clause_{\neg P}(X_1, \ldots, X_n) \end{cases}$$

2. Le méta-interprète
   Il faut coder toutes les résolutions possibles entre des clauses de longueur inférieure ou égale à n et dont la résolvante est elle aussi de longueur inférieure à n. Pour $n = 3$, on ajoutera par exemple

   $$clause_{P,Q,R}(X, Y, Z) \wedge clause_{S,\neg P}(V, X) \rightarrow clause_{Q,R,S}(Y, Z, V)$$

   pour coder qu'à partir de $P(X) \vee Q(Y) \vee R(Z)$ et $S(V) \vee \neg P(X)$ on peut obtenir par résolution $Q(Y) \vee R(Z) \vee S(V)$.
   L'ensemble des résolutions qu'il faut ainsi coder est certes énorme, mais fini puisque l'on se restreint à des clauses de longueur bornée par n.
   On code de la même manière toutes les factorisations possibles sur des clauses de longueur au plus n. Pour $n = 3$, cela donne par exemple

   $$clause_{P,P}(X, Y, X, Y) \rightarrow clause_P(X, Y)$$

   Cette règle permet bien d'effectuer la factorisation de $P(a, X) \vee P(Y, b)$ pour produire $P(a, b)$.
   L'ensemble des factorisations qu'il faut ainsi représenter est bien évidemment fini.

3. La restitution
   La restitution est très facile à effectuer puisqu'il s'agit de l'opération inverse de la traduction des faits. Il suffit d'écrire pour chaque prédicat $P(X_1, \ldots X_n)$ de la base

$$\begin{cases} clause_P(X_1, \ldots, X_n) \rightarrow P(X_1, \ldots, X_n) \\ clause_{\neg P}(X_1, \ldots, X_n) \rightarrow \neg P(X_1, \ldots, X_n) \end{cases}$$

Le méta interprète que l'on obtient réalise un achèvement complet au moins pour certaines bases. Contrairement au méta-interprète prolog, sa taille est exponentielle. On peut également noter que la technique employée pour ce méta-interprète aurait aussi bien pu être utilisée dans le cas prolog.

Chapitre 14

Les champs de production

Dans ce chapitre, nous étudions comment on peut utiliser les champs de production pour réaliser une forme d'achèvement partiel.

14.1 Une forme d'achèvement partiel

On a vu que pour achever complètement une base prolog ou datalog en conservant l'équivalence logique habituelle, il est parfois nécessaire d'ajouter un nombre infini de clauses. On peut cependant se demander si, en se limitant à un achèvement partiel, il n'est pas possible de se restreindre à l'ajout d'un nombre fini de clauses. Certes, on peut considérer que l'ajout à une base d'une seule de ses conséquences incalculable par chaînage avant forme un achèvement partiel. Toutefois, un tel achèvement est de peu d'intérêt car on ne sait pas caractériser pour quelles bases de faits le chaînage avant sera complet. Il nous importe donc de fournir une caractérisation des bases de faits pour lesquelles le calcul sera complet et s'assurer que ce critère a un sens dans la pratique.

La méthode que nous présentons maintenant se fonde sur la remarque suivante. Lorsqu'il est nécessaire d'ajouter un nombre infini de clauses pour effectuer un achèvement total, c'est soit parce que le domaine de Herbrand est infini (cas prolog), soit parce que le domaine de Herbrand peut être augmenté de manière arbitraire par ajout de faits (cas datalog et prolog). Plus précisément, le problème posé par le dernier cas est la présence de chaînes de faits qui permettent de simuler des fonctions et ainsi, des termes de taille quelconque.

Définition 148:

Une **chaîne de littéraux** est un ensemble de littéraux d'arité supérieure ou égale à deux qui peuvent être ordonnés en une séquence $(L_1, L_2, \ldots L_n)$ telle que L_i et L_{i+1} aient au moins un argument (variable ou constante) commun.

Une **chaîne de littéraux de longueur** n est une chaîne de littéraux comportant n littéraux.

Une **chaîne de faits** est une chaîne de littéraux.

Cependant, dans la pratique, il est bien rare d'effectuer des raisonnements sur des termes de taille quelconque. Par exemple, lorsque l'on raisonne sur la relation de parenté, on s'intéresse souvent aux parents et grand-parents mais l'on va rarement plus loin, en particulier dans les bases de données Datalog. Lorsqu'on souhaite achever la base codant le jeu

du démineur, on souhaite obtenir une base qui permette de jouer sur un damier 100*100 voire 10000*10000. Cependant, on ne souhaite certainement pas jouer sur un damier de coté 2^{10000} ! Il apparaît donc vain d'ajouter un nombre infini de clauses pour effectuer un achèvement total si la plupart des clauses ajoutées ne seront jamais utilisées au cours des inférences effectuées dans la pratique.

On considère donc que restreindre dans la base de faits la profondeur des termes (ou de manière équivalente en datalog la longueur des chaînes de faits) est compatible avec l'usage courant d'une base. On étudie alors les conséquences de cette restriction pour l'achèvement.

14.2 Le cas prolog

Malheureusement, limiter la profondeur des termes dans les faits ajoutés à une base prolog a peu d'intérêt pour l'achèvement. En effet, même quand les faits ne contiennent que des termes de profondeur inférieure à k, il peut être nécessaire de passer par un terme de profondeur supérieure à k pour inférer un autre fait dont tous les termes sont de profondeur inférieure à k. On est alors incapable de supprimer parmi les clauses ajoutées par un achèvement total celles qui contiennent de trop gros termes.

L'exemple suivant montre comment cela est possible. Soit $\mathcal{B}$ la base

$$\begin{cases} ①Power_k(f^2(0), f^{2^k}(0)). \\ ②P(f(X)) \rightarrow P(f^2(X)) \\ ③Power_k(f^2(0), X) \wedge P(X) \rightarrow P(g(X)) \\ ④P(g(f(X))) \rightarrow P(g(X)) \\ ⑤P(g(X)) \rightarrow P(X) \end{cases}$$

Cette base est construite pour que $\mathcal{B}, P(f(0)) \models P(0)$. En effet, on peut prouver $P(f^{2^k}(0)$ à partir de $P(f(0))$ par usage répété de la clause ②. On peut alors utiliser les clauses ③ et ① puis 2^k fois la clause ④ pour obtenir $P(g(0))$. Enfin, par la clause ⑤ on sait prouver $P(0)$.

On voit bien cependant qu'il est nécessaire de produire le fait $P(f^{2^k}(0)$ pour ensuite obtenir $P(0)$ à partir de $P(f(0))$ qui sont tous deux de profondeur inférieure à k.

Cet exemple peut laisser songeur puisque la première clause contient un terme de profondeur supérieure à k. Suffirait-il de limiter également la profondeur des termes de la base intentionnelle ? Hélas non. En effet, le prédicat $Power_k(X, Y)$ peut être défini facilement à partir des prédicats $Mul(X, Y, Z)$ et $Add(X, Y, Z)$ codant respectivement la multiplication et l'addition et dont les termes sont de profondeur au plus égale à 1.

14.3 Le cas datalog

Se restreindre en datalog à des bases de faits qui ne contiennent pas de chaîne de littéraux de longueur supérieure à k permet d'obtenir un achèvement partiel fini et en un temps fini de surcroît. Cela peut sembler paradoxal puisque l'exemple précédent montre que la limitation de la profondeur des termes en prolog ne permet pas de rendre fini l'ensemble des impliqués à ajouter. Comment cela se peut-il puisqu'il existe un moyen de coder les fonctions en datalog, ce qui devrait permettre d'aboutir au même problème ?

En fait, s'il est effectivement possible de transcrire la notion de fonction en datalog, il ne s'agit nullement d'une traduction fidèle et le pouvoir de calcul obtenu n'est pas le même. La transcription ne peut être fidèle que si la base de faits contient toutes les relations représentées par la fonction. Or, cela est interdit par la restriction sur la longueur des chaînes dans la base de faits.

Pour illustrer cela, nous reprenons l'exemple précédent en le traduisant en datalog. Soit $\mathcal{B}$ la base

$$\begin{cases} ①F(0,X_1) \wedge F(X_1,X_2) \wedge \ldots \wedge F(X_{2^k-1},X_{2^k}) \rightarrow Power_k(X_2,X_{2^k}). \\ ②F(X_0,X_1) \wedge F(X_1,X_2) \wedge P(X_1) \rightarrow P(X_2) \\ ③F(0,X_1) \wedge F(X_1,X_2) \wedge Power_k(X_2,X_3) \wedge P(X_3) \wedge G(X_3,X_4) \rightarrow P(X_4) \\ ④F(X_0,X_1) \wedge G(X_1,X_2) \wedge G(X_0,X_3) \wedge P(X_2) \rightarrow P(X_3) \\ ⑤G(X_0,X_1) \wedge P(X_1) \rightarrow P(X_0) \end{cases}$$

Par résolutions avec la clause ② et factorisations successives, on peut obtenir

$$F(X_0,X_1) \wedge F(X_1,X_2) \wedge \ldots F(X_{2^k-1},X_{2^k}) \wedge P(X_1) \rightarrow P(X_{2^k})$$

On peut alors résoudre et factoriser avec les clauses ① et ③ pour obtenir

$$F(0,X_1) \wedge F(X_1,X_2) \wedge \ldots F(X_{2^k-1},X_{2^k}) \wedge G(X_{2^k},Y_{2^k}) \wedge P(X_1) \rightarrow P(Y_{2^k})$$

À ce moment, par résolutions avec ④ et factorisations on peut prouver

$$\begin{aligned} &F(0,X_1) \wedge F(X_1,X_2) \wedge \ldots \wedge F(X_{2^k-1},X_{2^k}) \wedge \\ &\quad G(0,Y_0) \wedge G(X_1,Y_1) \wedge G(X_2,Y_2) \wedge \ldots G(X_{2^k},Y_{2^k}) \wedge P(X_1) \\ &\quad \rightarrow P(Y_0) \end{aligned}$$

Enfin, en utilisant la clause ⑤ on obtient

$$\begin{aligned} &F(0,X_1) \wedge F(X_1,X_2) \wedge \ldots F(X_{2^k-1},X_{2^k}) \wedge \\ &\quad G(0,Y_0) \wedge G(X_1,Y_1) \wedge G(X_2,Y_2) \wedge \ldots G(X_{2^k},Y_{2^k}) \wedge P(X_1) \\ &\quad \rightarrow P(0) \end{aligned}$$

Cette dernière clause représente l'implication $P(f(0)) \rightarrow P(0)$. Cependant, il est impossible d'éliminer les littéraux $F(X,Y)$ et $G(X,Y)$ qui y figurent pour obtenir $F(0,X) \wedge P(X) \rightarrow P(0)$. En effet, il est indispensable au cours du raisonnement de savoir que $f(0)$ a un successeur $f^2(0)$, de même que $f^2(0)$ a un successeur $f^3(0)$ et ainsi de suite. Cette existence est acquise en prolog, mais pas en datalog et c'est bien là ce qui fait toute la différence. En effet, la clause que nous avons obtenue est inutilisable si la base de faits ne contient pas de chaîne de faits de longueur supérieure à k. De ce fait, il est inutile de l'ajouter pour effectuer l'achèvement partiel.

On est alors tenté de croire qu'en interdisant les chaînes de longueur supérieure à k dans la base de faits, les inférences réalisées par chaînage avant sur la base compilée seront complètes. Hélas, il n'en est rien. En effet, à partir d'un ensemble de faits qui ne contient aucune chaîne, il est possible d'inférer de nouveaux faits qui permettront de créer une chaîne, comme le montre cet exemple :

Soit la base

$$\begin{cases} R(X_1, X_2) \wedge R(X_3, X_4) \rightarrow R(X_4, X_2) \\ R(X_1, X_2) \wedge R(X_3, X_4) \wedge R(X_4, X_2) \rightarrow R(X_2, X_3) \\ R(X_1, X_2) \wedge R(X_3, X_4) \wedge R(X_4, X_2) \wedge R(X_2, X_3) \rightarrow R(X_3, X_1) \\ R(X_1, X_2) \wedge R(X_3, X_4) \wedge R(X_4, X_2) \wedge R(X_2, X_3) \wedge R(X_3, X_1) \rightarrow R(X_1, X_4) \end{cases}$$

À partir de la base de faits $\{R(1,2), R(3,4)\}$ qui ne contient pas de chaîne de littéraux, le chaînage avant va produire $R(4,2)$ puis $R(2,3)$, $R(3,1)$ et enfin $R(1,4)$. Les faits obtenus forment une chaîne de littéraux de longueur 6. En ajoutant d'autre faits $R(2n, 2n+1)$ on peut obtenir in fine des chaînes de littéraux de longueur arbitrairement grande. En effet, ce programme calcule la relation correspondant à un graphe complet. Puisque le graphe est complet, il est possible d'obtenir une chaîne de longueur $n(n-1)/2$ à partir d'une base de faits comportant n symboles de constantes.

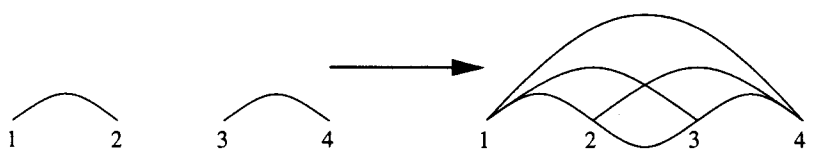


Figure 56: Obtention d'une chaîne à partir de faits sans chaîne

On peut cependant douter de l'intérêt pratique d'un tel programme. En fait, on peut même conjecturer qu'aucun programme datalog intéressant ne pose ce type de problème.

Dans tous les cas, il est possible de vérifier a posteriori que les inférences effectuées par le chaînage avant ont été complètes. En effet, notre achèvement partiel est valide si la base de faits ne contient pas de chaîne de longueur supérieure à k. Si cette condition est vérifiée à la fin des déductions, alors, nous avons la certitude que les inférences ont été complètes. Si le critère n'est pas vérifié, il y a un doute. Si donc il est difficile de prévoir pour quelles bases le calcul est complet, il est en revanche simple de vérifier que tout a été déduit. Cette approche semble donc pleinement satisfaisante dans la pratique.

L'idée de la méthode est celle-ci. Lorsqu'on suppose que la base de faits ne contient pas de chaînes de longueur supérieure à k, on peut éviter de produire des règles dont la condition contient une chaîne de longueur supérieure à k puisqu'on sait que cette condition ne sera jamais vérifiée. Transcrite en termes de clauses, cette idée s'exprime formellement ainsi :

Proposition 149:

Une clause comportant une chaîne de littéraux de longueur $k+1$ et un littéral l est inutilisable par le chaînage avant pour prouver l si la base de faits ne contient aucune chaîne de faits de longueur strictement supérieure à k.

Démonstration 150:

Pour que le chaînage avant puisse utiliser la clause pour prouver l, il faut que tous ses littéraux sauf l soient présents sous forme négative et correctement instanciées dans

la base de faits. Or, toute instance d'une chaîne est une chaîne de même longueur. Il faudrait donc que la base de faits contienne une chaîne de longueur $k+1$ ce qui contredit l'hypothèse.

Il reste alors à montrer que l'ensemble des impliqués premiers ne contenant pas de chaîne de longueur supérieure à $k+1$ est fini (la limite de $k+1$ provient de ce que le littéral l de la démonstration précédente peut augmenter la taille de la chaîne de une unité). Pour ce faire, on montre d'abord que l'on ne peut construire qu'un ensemble fini de chaînes de longueur inférieure ou égale à $k+1$.

Proposition 151:

En datalog, l'ensemble des chaînes de littéraux de longueur inférieure ou égale à $k+1$ est fini après simplification par w-subsomption.

Démonstration 152:

Soit n le nombre de constantes de la base et a l'arité maximale des prédicats de la base. Une chaîne de littéraux de longueur inférieure ou égale à $k+1$ contient au plus $(k+1).a$ positions d'arguments de prédicat. Une chaîne peut donc contenir au plus $(k+1).a$ variables différentes. De ce fait, il y a au plus $1+n+(k+1).a$ choix pour remplir une position d'argument (le 1 provient de ce que la position peut être laissée vide si la longueur de la chaîne est inférieure à $k+1$). Cela fait donc au plus $(1+n+(k+1).a)^{(k+1).a}$ séquences possibles différentes (à un renommage des variables près) d'arguments dans une chaîne de littéraux de longueur inférieure ou égale à $k+1$. Il reste à attribuer chaque argument de ces séquences à un symbole de prédicat pour obtenir une chaîne. Soit p le nombre de symboles de prédicats. Le nombre maximal de chaînes est obtenu quand on considère que tous les prédicats sont d'arité 1. Dans ce cas, il faut choisir pour chaque argument à quel prédicat il appartient. On obtient donc $(p(1+n+(k+1).a))^{(k+1).a}$ comme nombre maximal de chaînes de littéraux de longueur inférieure à $k+1$. Ce nombre est fini.

Enfin, on montre que, puisqu'une clause ne peut contenir qu'un nombre fini de chaînes de longueur inférieure ou égale à $k+1$, l'ensemble des clauses sans chaîne plus longue que $k+1$ est fini.

Proposition 153:

L'ensemble des clauses datalog qui ne comportent pas de chaîne de littéraux de longueur supérieure à $k+1$ est fini après simplification par w-subsomption.

Démonstration 154:

Une clause peut contenir plusieurs chaînes de littéraux de longueur inférieure ou égale à $k+1$. On a cependant vu qu'il y avait moins de $(p(1+n+(k+1).a))^{(k+1).a}$ de ces chaînes. Donc, une clause non w-subsumée qui ne contient pas de chaînes de longueur strictement supérieure à $k+1$ contient au plus $(p(1+n+(k+1).a))^{(k+1).a}$ chaînes. Cette clause contient alors au plus $(k+1).a.(p.(1+n+(k+1).a))^{(k+1).a}$ positions d'arguments dans une chaîne qui peuvent être remplies d'au plus $1+n+(k+1).a.(p(1+n+(k+1).a))^{(k+1).a}$ manières différentes (à un renommage près). En reprenant les mêmes arguments que dans la démonstration précédente, on déduit que le nombre de clauses possibles est fini.

Il reste maintenant à fournir un algorithme de production des impliqués premiers sans chaîne plus longue que $k+1$.

14.4 La SOL-résolution

La SOL-résolution proposée par [Ino92] est un bon candidat pour effectuer le calcul précédent. Il s'agit d'une extension au premier ordre des champs de production de [Sie87] basée sur l'OL résolution de [CL73]. L'idée des algorithmes utilisant les champs de production est de ne produire qu'un sous-ensemble des impliqués premiers possédant une certaine caractéristique, appelée champ de production. Le champ de production que nous utiliserons sera donc «ne contient pas de chaîne de longueur strictement supérieure à $k+1$». Nous détaillons maintenant le fonctionnement de la SOL-résolution tel que décrit par [Ino92].

La SOL-résolution utilise la notion de résolution ordonnée.

Définition 155:

Un **littéral encadré** (**framed literal**) est un littéral marqué. On le note $\boxed{l}$. Ces littéraux sont aussi appelés **littéraux ancêtres**.

Une **clause ordonnée** est une séquence de littéraux ou de littéraux encadrés. On la note $\vec{C}$. Les littéraux encadrés d'une clause n'ont pas de signification sémantique (il s'agit de littéraux fantômes). L'ordre des littéraux ne modifie pas non plus la sémantique de la clause.

Définition 156:

Soient deux clauses ordonnées $\vec{C_1}$ et $\vec{C_2}$ qui ne partagent pas de variables et telles qu'il existe un littéral P vérifiant $P \in \vec{C_1}$, $\neg P \in \vec{C_2}$ et $P, \neg P$ non encadrés.

Si P et $\neg P$ ont un m.g.u. σ, la **résolution binaire ordonnée** de $\vec{C_1}$ et $\vec{C_2}$ est la clause ordonnée $\sigma((\vec{C_1} - \{P\}).\boxed{\neg P}.(\vec{C_2} - \{\neg P\}))$, le symbole . représentant la concaténation des clauses ordonnées. Les littéraux $(\vec{C_2} - \{\neg P\})$ sont appelés **frères** du littéral encadrés $\boxed{\neg P}$.

Les littéraux encadrés ont pour but de témoigner de la présence dans la clause ordonnés de leurs frères, ainsi que de conserver la trace des substitutions qui leur sont appliqués. Ils doivent donc disparaître dès que leurs frères risquent d'être éliminés de la clause. On utilise pour cela la fonction $trunc$.

Définition 157:

Soit $\boldsymbol{trunc}$ la fonction qui à une clause ordonnée $\vec{C}$ associe la clause ordonnée obtenue à partir de $\vec{C}$ en supprimant tous les littéraux encadrés qui ne sont pas précédés par un littéral normal.

Les littéraux encadrés permettent d'effectuer une résolution sur ancêtre.

Définition 158:

Soit une clause ordonnée $\vec{C}$ contenant un littéral non encadré l et un littéral encadré $\boxed{\neg l}$ figurant après l tels que l et $\neg l$ ont pour m.g.u. σ. La **résolution sur ancêtre** consiste à produire la clause ordonnée $trunc(\sigma(\vec{C} - \{l\}))$.

Cette résolution sur ancêtre revient à faire une résolution avec la précédente clause centrale composée du littéral encadré et de ses frères.

Une restriction supplémentaire est de n'autoriser à effectuer de résolution qu'avec le premier littéral de la clause comme pivot. On introduit alors la fonction $right$.

Définition 159:

Soit ***right*** la fonction qui associe à une clause ordonnée la même clause privée de son premier littéral.

La dernière notion à introduire pour présenter la SOL-résolution est celle de clause structurée.

Définition 160:

Une **clause structurée** est la paire formée d'une clause P et d'une clause ordonnée $\vec{Q}$. Elle est notée $\left\langle P, \vec{Q} \right\rangle$.

La clause P de la clause structurée est la partie qui appartient au champ de production et que l'on ne souhaite plus modifier. La clause $\vec{Q}$ quand à elle représente la partie de la clause que l'on souhaite éliminer.

La SOL-résolution permet de calculer les impliqués premiers appartenant à un champ de production $Prod$ qui sont issus de l'ajout d'une clause C à la base $\mathcal{B}$, c'est à dire, l'ensemble des impliqués premiers I tels que $(I \in Prod) \wedge (\mathcal{B} \not\models I) \wedge (\mathcal{B} \cup C \models I)$.

L'algorithme consiste à construire une dérivation représentée par une séquence de clauses structurées $D_i = \left\langle P_i, \vec{Q_i} \right\rangle$ obéissant aux règles ci-dessous [Ino92]

1. $D_0 =< \square, \vec{C} >$
2. $D_n =< S, \square >$
3. Pour chaque $D_i = \left\langle P_i, \vec{Q_i} \right\rangle$, $P_i \cup Q_i$ n'est pas une tautologie.
4. Pour chaque $D_i = \left\langle P_i, \vec{Q_i} \right\rangle$, Q_i n'est pas subsumé par un Q_j $(j < i)$ avec la substitution identité.
5. Pour chaque $D_i = \left\langle P_i, \vec{Q_i} \right\rangle$, P_i appartient au champ de production.
6. $D_{i+1} = \left\langle P_{i+1}, \vec{Q_{i+1}} \right\rangle$ est généré à partir de $D_i = \left\langle P_i, \vec{Q_i} \right\rangle$ par l'usage d'une des trois règles suivantes au choix :
   (a) **Skip**
   Soit l le littéral le plus à gauche de $\vec{Q_i}$. Si $P_i \cup \{l\}$ appartient au champ de production
   $P_{i+1} = P_i \cup \{l\}$
   $\vec{Q_{i+1}} = trunc(right(\vec{Q_i}))$
   (b) **Resolve**
   Soit l le littéral le plus à gauche de $\vec{Q_i}$
   Soit D une clause de $\mathcal{B} \cup C$ contenant un littéral $\neg k$ tel que k et l sont unifiables avec σ comme m.g.u.
   $P_{i+1} = \sigma(P_i)$
   $\vec{Q_{i+1}} = trunc(\sigma(D - \{\neg k\}.\boxed{1}.right(\vec{Q_i})))$
   (c) **Reduce**
   Soit l le littéral le plus à gauche de $\vec{Q_i}$
   On applique au choix
      i. Si P_i ou $\vec{Q_i}$ contient un littéral k non encadré qui est soit différent de l (*factoring*) soit une autre occurrence de l (*merge*), et tel que l et k sont unifiables avec σ pour m.g.u.

$P_{i+1} = \sigma(P_i)$
$\vec{Q_{i+1}} = trunc(right(\sigma(\vec{Q_i})))$

ii. Si $\vec{Q_i}$ contient un littéral encadré $\boxed{\neg k}$ (*ancestry*) tel que l et k sont unifiables avec σ pour m.g.u.
$P_{i+1} = \sigma(P_i)$
$\vec{Q_{i+1}} = trunc(right(\sigma(\vec{Q_i})))$

On dit alors que la clause S est produite par SOL-résolution à partir de $\mathcal{B} \cup C$. Pour produire tous les impliqués premiers d'une base appartenant au champ de production, il suffit d'utiliser l'algorithme suivant :

1: $\mathcal{B}' \leftarrow \varnothing$
2: **for all** $C \in \mathcal{B}$ **do**
3: calculer les clauses produites par SOL-résolution à partir de $\mathcal{B}' \cup C$
4: $\mathcal{B}' \leftarrow \mathcal{B}' \cup C$
5: **end for**

Il reste à montrer que cet algorithme termine pour le champ de production qui nous intéresse.

Proposition 161:

La SOL-résolution termine en datalog (tout en restant complète) pour le champ de production «ne comporte pas de chaîne de littéraux de longueur supérieure à $k + 1$» à condition d'imposer la condition $P_i \cup Q_i$ ne contient pas de chaîne de longueur strictement supérieure à $k + 2$ et de choisir un ordre particulier pour les littéraux d'une clause ordonnée.

Démonstration 162:

On montre qu'on peut imposer à tout moment dans la SOL-résolution la condition $P_i \cup Q_i$ ne contient pas de chaîne de longueur strictement supérieure à $k + 2$ sans risquer de perdre un impliqué sans chaîne de longueur strictement supérieure à $k + 1$. Le nombre $k+2$ s'explique par le fait qu'il faille pouvoir accepter de manière temporaire des chaînes de longueur $k + 2$ lorsque le dernier maillon de la chaîne sera détruit immédiatement.

En ignorant provisoirement le fait que les clauses soient ordonnées, supposons qu'à un instant donné $P_i \cup Q_i$ contienne une chaîne de longueur supérieure ou égale à $k + 3$. L'un au moins des éléments de cette chaîne doit être supprimé ultérieurement pour satisfaire le champ de production. Cette suppression doit avoir lieu soit par factorisation avec un autre élément de $P_i \cup Q_i$, soit par résolution binaire. Or, il est possible de remonter les opérations de factorisation et de résolution binaire (cf. propositions 131 et 129) pour effectuer cette suppression au plus tôt. Cela signifie qu'il existe une autre branche de l'arbre SOL qui vérifie la contrainte et fournit le même résultat. Il est donc inutile d'explorer plus avant cette branche.

Quand on utilise des clauses ordonnées, il faut choisir l'ordre des littéraux lors de la création d'une nouvelle résolvante de manière à ce que les prédicats qui forment des chaînes de longueur $k + 2$ apparaissent le plus à gauche possible. De la sorte, il seront les premiers à être éliminés.

Comme l'ensemble des clauses sans chaîne de longueur supérieure ou égale à $k + 3$ est fini, on en déduit que la SOL-résolution termine nécessairement.

Chapitre 15

Les cycles

La méthode d'achèvement par cycles semble être la meilleure candidate pour étendre l'achèvement au premier ordre. Une première raison est qu'il s'agit d'une méthode syntaxique basée sur la résolution. Il existe certes des méthodes d'arbre sémantique pour le premier ordre [Lep91], mais pas aussi largement utilisées que la résolution. D'autre part, ces méthodes semblent difficilement pouvoir s'accommoder d'un nombre infini d'impliqués premiers. En revanche, l'usage de cycles permet d'expliquer finement pourquoi il faut parfois ajouter un nombre infini de clauses et comment représenter de manière éventuellement finie cet ensemble.

15.1 Représentation sous forme de graphe

Il faut d'abord adapter le graphe utilisé dans la méthode des cycles propositionnelle. La principale caractéristique de ce graphe doit être de permettre la détection des fusions par la recherche de cycles. Comme des littéraux comme $P(a, X)$ et $P(X, b)$ peuvent être fusionnés, ils doivent correspondre à un seul et même nœud sur le graphe. Les nœuds du graphe représenteront alors les formes les plus générales des prédicats, en l'occurrence $P(X, Y)$. Comme en calcul propositionnel, une clause est représentée par un nœud qui l'identifie. Les littéraux composant la clause lui sont reliés par un arc sur lequel est précisé comment la forme générale du littéral doit être instanciée pour figurer dans la clause. Il est nécessaire de faire porter cette substitution par l'arête et non par le nœud clause car un littéral peut apparaître plusieurs fois dans une même clause sous différentes instances (ex : $P(a, X) \vee P(Y, b)$).

On aboutit alors à la définition suivante :

Définition 163:

Le **graphe général** associé à une base du premier ordre est le multigraphe[11] tel que

- l'ensemble des nœuds est constitué par
  - l'ensemble des littéraux les plus généraux de la base et leurs négations (**nœud littéral**)
  - l'ensemble des identificateurs de clauses de la base (**nœud clause**)

[11] Un multigraphe est un graphe tel qu'il peut y avoir plus d'une arête entre deux nœuds. Les graphes que nous considérons maintenant ne sont plus des 1-graphes.

- l'ensemble des arêtes est constitué par
  - l'ensemble des L–$\neg L$ pour tout L littéral le plus général de la base (**arête pivot**). Toute arête pivot est implicitement étiquetée par la condition permettant d'unifier les littéraux qu'elle relie.
  - l'ensemble des arêtes C–L étiquetées par la substitution σ si L est un nœud littéral, C un nœud clause et $\sigma(L)$ appartient à la clause C (**arête clause**).

Par convention, on note les instances les plus générales des littéraux en utilisant les symboles $@_1, @_2, \ldots$ pour figurer les variables ($@_n$ signifiant *argument numéro* n). On représentera donc la clause $P(X_1, X_2, \ldots, X_n) \vee Q(Y_1, Y_2, \ldots, Y_m)$ par le graphe général :

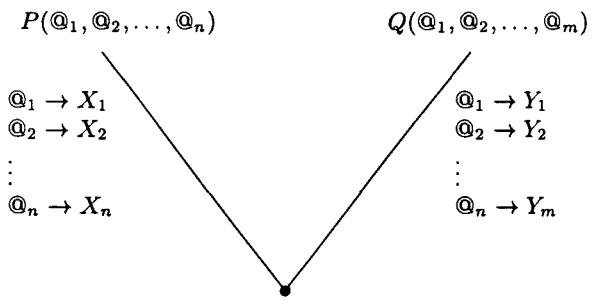


Figure 57: Graphe général codant la clause $P(X_1, X_2, \ldots, X_n) \vee Q(Y_1, Y_2, \ldots, Y_m)$

Une arête pivot est représentée comme ceci :

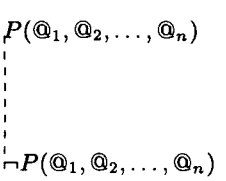


Figure 58: Représentation d'un littéral et de son opposé.

Enfin, le graphe général correspondant à la base

$$\begin{cases} ① \; P(a, X) \vee P(Y, b) \\ ② \; \neg P(X, Y) \vee Q(X, Y) \\ ③ \; Q(b, Z) \end{cases}$$

est le suivant

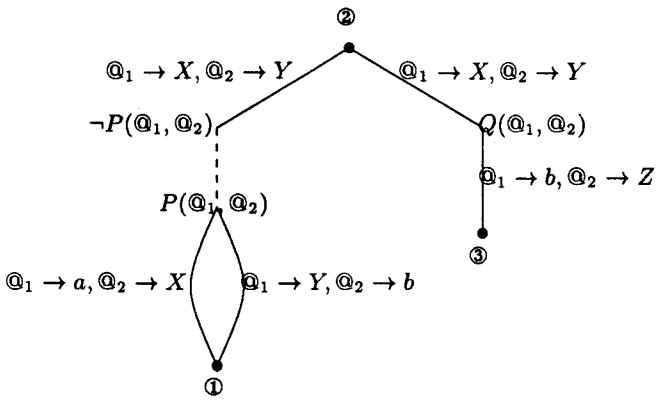


Figure 59: Exemple de graphe général.

On peut aussi construire un graphe sans exiger que les nœuds littéraux représentent les littéraux les plus généraux. Dans ce cas, plusieurs instances d'un même littéral peuvent coexister dans le graphe.

Définition 164:

Le **graphe instancié** associé à une base du premier ordre est tel que

- l'ensemble des nœuds est constitué par
  - l'ensemble des littéraux de la base et leur négation (**nœud littéral**)
  - l'ensemble des identificateurs de clauses de la base (**nœud clause**)
- l'ensemble des arêtes est constitué par
  - l'ensemble des L–$\neg L$ pour tout L et $\neg L$ littéraux de la base pouvant s'unifier (**arête pivot**).
  - l'ensemble des C–L tel que C est un nœud clause et L appartient à la clause C (**arête clause**).

Un graphe instancié ressemble assez fortement à un graphe de connexions [Kow75]. En particulier, il ne permet plus une détection facile des fusions. Cependant, contrairement à la définition d'un graphe de connexions, un nœud littéral dans un graphe instancié peut être partagé par plusieurs clauses.

Exemple 165:

La base

$$\begin{cases} ① \; P(a,X) \vee P(Y,b) \\ ② \; \neg P(X,Y) \vee Q(X,Y) \\ ③ \; P(Y,b) \vee Q(b,Z) \end{cases}$$

possède le graphe instancié suivant

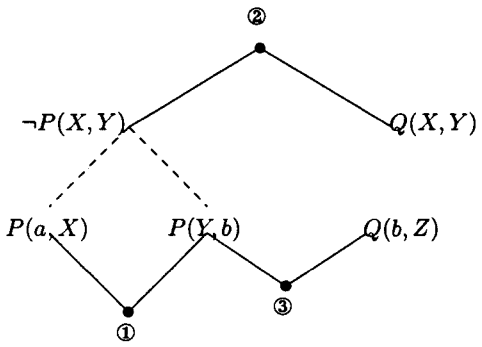


Figure 60: Exemple de graphe instancié.

Pour mieux visualiser certains phénomènes, il est parfois plus intéressant de travailler sur un graphe instancié correspondant à des instanciations des clauses d'une base $\mathcal{B}$ plutôt que de travailler sur le graphe général de $\mathcal{B}$. On dit dans ce cas que l'on a déplié le graphe général puisque l'on fait apparaître alors plusieurs occurrences des arêtes là où il n'y en avait qu'une, un peu comme si toute ces arêtes avaient été repliées pour se confondre en une seule dans le graphe général.

Définition 166:

On appelle **dépliage d'un graphe général** d'une base $\mathcal{B}$ le graphe instancié qui correspond à un ensemble d'instanciations de la base $\mathcal{B}$.

On peut donc faire apparaître sur ce dépliage plusieurs instances d'une même clause de $\mathcal{B}$. Un tel dépliage présente l'avantage d'éviter de considérer des cycles non-élémentaires et a surtout valeur d'exemple.

On définit enfin la notion de graphe clos qui correspond à un dépliage d'une base pour obtenir une base propositionnelle.

Définition 167:

Un **graphe clos** est un dépliage d'un graphe général qui ne contient pas de variable (tous les littéraux sont clos).

Bien sûr, on retrouve sur le graphe clos toutes les propriétés des cycles propositionnels.

15.2 Les cycles du graphes

Les cycles d'un graphe général qui trahissent d'éventuelles fusions sont de la forme

$$L\text{–}(C_i\text{–}L_i\text{-}\cdots\text{-}\neg L_i\text{–})^*C\text{–}L$$

Comme en théorie des langages, l'étoile représente la répétition un nombre quelconque de fois (éventuellement nul) d'un schéma. Les cycles suivants répondent donc à ce schéma général.

Cycle	Signification
L–C–L	factorisation d'une clause
L–C_1–L_1···$\neg L_1$–C_2–L	résolution avec fusion
L–C_1–L_1···$\neg L_1$–C_2–L_2···$\neg L_2$–C_3–L	résolution avec fusion
...	

La plupart des résultats obtenus en calcul propositionnel sur les cycles s'étendent au premier ordre. La différence majeure est qu'il n'est pas possible d'éliminer les cycles non élémentaires et qu'un cycle ambigu n'a plus la même caractérisation.

15.3 Présentation informelle

Nous introduisons maintenant une méthode permettant de gérer les cycles non élémentaires sur une série d'exemples de difficulté croissante.

15.3.1 Exemple d'achèvement (1)

Soit la connaissance suivante :

- $Entier(X) \wedge Successeur(X, XX) \rightarrow Entier(XX)$

  | *Si X est un entier et XX est son successeur, alors, XX est aussi un entier.*

- $MemoireBonne \rightarrow Entier(0)$

  | *Si ma mémoire est bonne, 0 est un entier.*

- $MemoireBonne \rightarrow \neg Entier(4)$

  | *En revanche, pour autant que je m'en souvienne, 4 n'est pas un entier.*

Bien évidemment, il faut savoir déduire que ma mémoire me trahit une fois les relations de succession entre entiers insérées dans la base de faits.

Bien sûr, ajouter à la base la clause

$\neg M \vee \neg S(0, X_1) \vee \neg S(X_1, X_2) \vee \neg S(X_2, X_3) \vee \neg S(X_3, 4)$

permet de déduire que ma mémoire est mauvaise à partir des faits

$$S(0,1), S(1,2), S(2,3), S(3,4)$$

Cela ne constitue cependant pas un achèvement total. En effet, les faits

$$S(0,a), S(a,b), S(b,c), S(c,d), S(d,4)$$

impliquent logiquement que ma mémoire est mauvaise mais la clause ajoutée ne permet pas cette démonstration. En fait, un achèvement total de cette base suivant cette méthode nécessite l'ajout d'un nombre infini de clauses.

Pour éviter ce problème, on peut utiliser une autre méthode que nous détaillons sur cet exemple.

Le graphe général de cette base est le suivant :

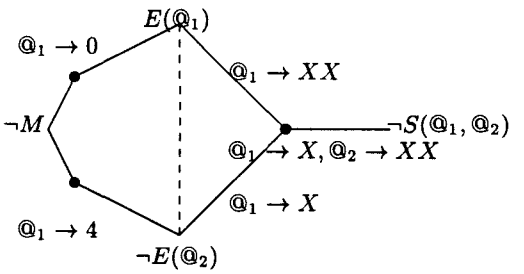


Figure 61: Graphe général de la base

Voici le dépliage qui nous intéresse :

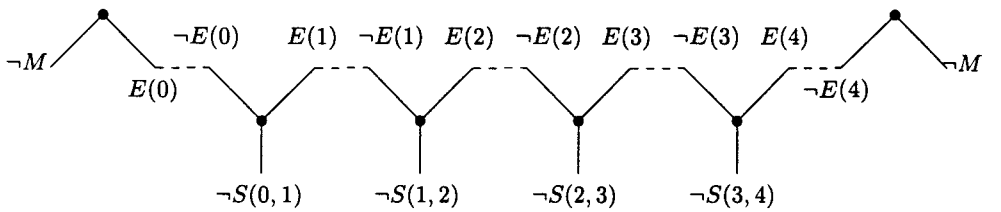


Figure 62: Dépliage de la base

Supposons que les faits $S(0,1), S(1,2), S(2,3), S(3,4)$ figurent tous dans la base de faits. Pour que la base soit achevée, il faudrait pouvoir produire $\neg M$ par chaînage avant. Or, cette déduction est bloquée à chaque extrémité de la chaîne par la présence du littéral à prouver.

Il faudrait débloquer l'une des extrémités de la chaîne. Pour ce faire, supposons que $\neg M$ soit faux. On peut alors utiliser l'extrémité gauche de la chaîne ($M \to E(0)$) pour prouver $E(0)$, puis, en remontant la chaîne, prouver $E(1), E(2), E(3)$ et $E(4)$. On peut alors prouver $\neg M$.

Ce débloquage correspond en fait à un raisonnement par l'absurde. Nous avons supposé que $\neg M$ était faux, pour en déduire que cela implique que $\neg M$ est vrai. De ce fait, $\neg M$ ne peut être faux.

On ne peut certes pas briser un cycle sans précaution. En effet ajouter directement $E(0)$ (le prédicat généré par le débloquage) à la base changerait la sémantique de la base. En fait, le déblocage consiste à ajouter $E(0)$ *sous la condition M est vrai*. Nous noterons cela en disant qu'on ajoute $\boldsymbol{E(0|M)}$ qui se lit comme une probabilité conditionnelle, à savoir : $E(0)$ sachant M. La notation $E(0|M)$ représente bien un prédicat puisqu'on aurait aussi bien pu utiliser la notation $E_M(0)$. La notation conditionnelle a le mérite d'être claire et de bien marquer le fait qu'il s'agit d'un nouveau symbole de prédicat. En effet, cette méthode d'achèvement étend le vocabulaire de la base.

Donc, débloquer le cycle consiste dans ce cas à ajouter la clause $E(0|M)$. On appelle cette opération **initiation**. Il faut ensuite pouvoir propager cette information conditionnelle, c'est à dire spécifier que si 0 est un entier sachant M et que le successeur de 0 est 1, alors, 1 est un entier sachant M, et ainsi de suite pour toute relation de succession. On ajoutera donc la clause $E(X|M) \wedge S(X, XX) \to E(XX|M)$. On dit que cette clause effectue une **propagation** de l'information conditionnelle. Enfin, cette propagation doit se terminer et permettre la déduction du fait $\neg M$. On ajoute dans ce but la clause $E(4|M) \to \neg M$. Il s'agit de l'étape de **terminaison**. Lorsqu'on ajoute ces clauses d'initiation, propagation et terminaison pour un cycle donné, on dit qu'on a **équipé** ce cycle.

Les clauses ajoutées à la base sont donc

$E(0\|M)$	initiation
$\neg E(X\|M) \vee \neg S(X,XX) \vee E(XX\|M)$	propagation
$\neg E(4\|M) \vee \neg M$	terminaison

Le dépliage de la base correspondant à la déduction qui nous intéresse est le suivant :

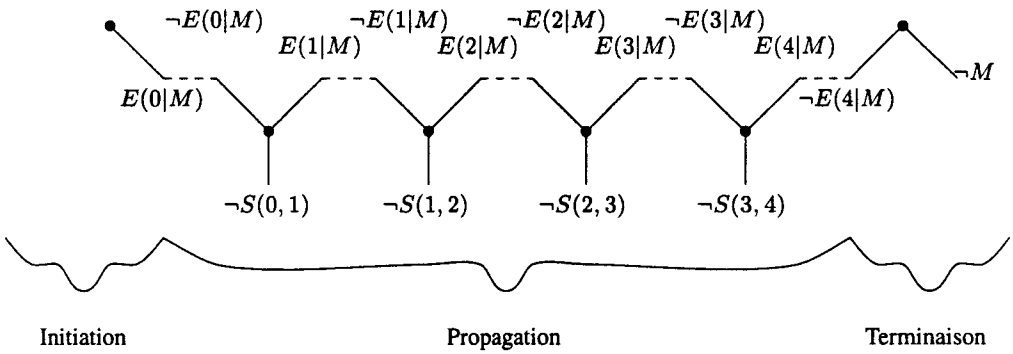


Figure 63: Phases de parcours d'un cycle

À partir des faits $S(0,1), S(1,2), S(2,3), S(3,4)$ et de la clause initiatrice $E(0|M)$, le chaînage avant sait prouver $\neg M$. On peut noter que la même déduction sera possible pour toute chaîne de relations de succession entre 0 et 4. D'autre part, la base obtenue est équivalente par ses impliqués à la base initiale.

En procédant de la sorte, il a donc été possible d'achever de manière finie cette base en obtenant une base équivalente par son comportement.

15.3.2 Exemple d'achèvement (2)

On étudie maintenant un exemple un peu plus compliqué puisque la clause de propagation contient une occurrence du littéral fusionné. La base est la suivante :

- $Normal \rightarrow (Entier(X) \wedge Successeur(X, XX) \rightarrow Entier(XX))$

  | *Si tout est normal (i.e. si l'on travaille bien sur des entiers), alors, si X est un entier et XX est son successeur, XX est aussi un entier.*

Une personne mésinterprète la signification des prédicats de cette base. Elle considère que $Entier(X)$ signifie «la personne X a un caractère entier» et $Successeur(X,Y)$ «la personne Y succède à X au poste de PDG». Elle introduit alors dans la base les faits suivants :

- $Entier(Jean)$

  | *Jean a un caractère entier.*

- $\neg Entier(Paul)$

  | *Ce n'est en revanche pas le cas de Paul.*

- $Successeur(Jean, Pierre)$

  | *Pierre succède à Jean au poste de PDG.*

- $Successeur(Pierre, Paul)$

  | *Le successeur de Pierre est Paul.*

Un système complet doit alors signaler l'erreur en produisant $\neg Normal$. En effet, supposons que $Normal$ soit vrai. Dans ce cas, le successeur d'un entier est un entier. Donc, Pierre doit avoir un caractère entier, de même que Paul. Or, ce n'est pas le cas. Il y a contradiction et l'on en déduit que $Normal$ est nécessairement faux.

Un tel raisonnement ne peut être effectué par le chaînage avant. En effet, parmi les deux instances de clause intéressantes, aucune ne peut produire de littéral par résolution unitaire, comme on peut le voir ci-dessous où les littéraux qui peuvent être effacés par résolution ont été soulignés.

$$\neg N \vee \underline{\neg E(Jean)} \vee \underline{\neg S(Jean, Pierre)} \vee E(Pierre)$$
$$\neg N \vee \underline{\neg E(Pierre)} \vee \underline{\neg S(Pierre, Paul)} \vee \underline{E(Paul)}$$

Il est donc nécessaire d'achever cette base. Voyons comment effectuer cela sur le graphe de la base.

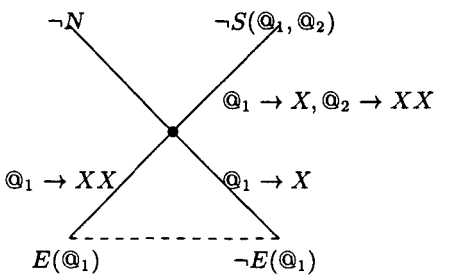


Figure 64: Graphe de la base

Prenons l'exemple suivant de cycle non élémentaire.

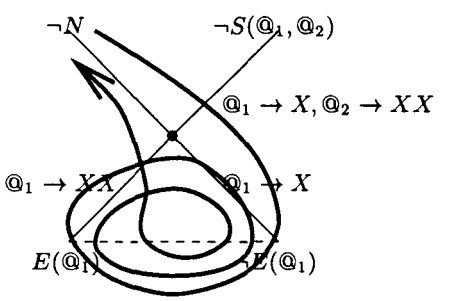


Figure 65: Exemple de cycle

On peut déplier ce cycle comme suit

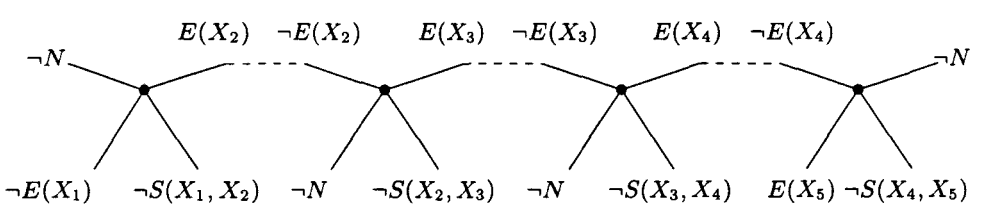


Figure 66: Dépliage du cycle

Si l'on fait l'hypothèse que N est vrai, on peut partiellement simplifier le dépliage comme suit :

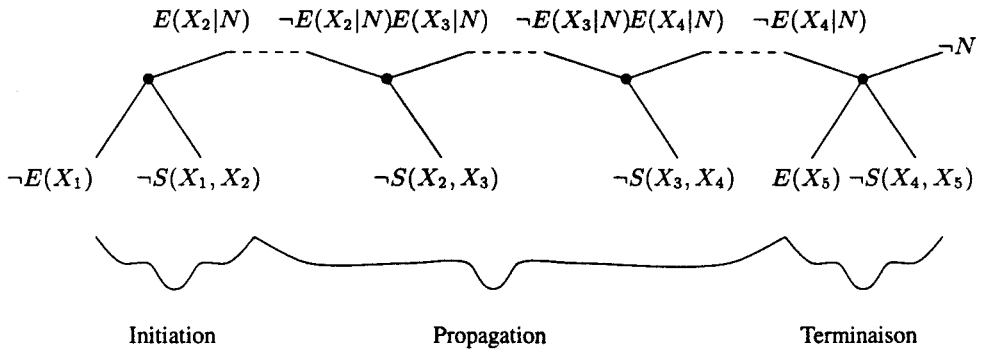


Figure 67: Simplification partielle du graphe déplié sous l'hypothèse N vrai

Chaque occurrence de $\neg N$ dans la clause de propagation a été simplifiée en tenant compte de l'hypothèse. Cette simplification permet de casser le cycle et de déduire N à partir des faits $\{E(1), S(1,2), S(2,3), S(3,4), S(4,5), \neg E(5)\}$.

On ajoute donc comme précédemment les clauses d'initiation, propagation et terminaison pour achever la base. On remarquera que dans ce cas, la clause d'initiation n'est pas unitaire.

$$\begin{array}{ll} \neg E(X) \vee \neg S(X, XX) \vee E(XX|N) & \text{initiation} \\ \neg E(X|N) \vee \neg S(X, XX) \vee E(XX|N) & \text{propagation} \\ \neg E(X|N) \vee \neg S(X, XX) \vee E(XX) \vee \neg N & \text{terminaison} \end{array}$$

15.3.3 Exemple d'achèvement (3)

Dans les deux exemples précédents, la tête de cycle ne comportait pas de variable. Ce n'est pas le cas de l'exemple suivant. On imagine trois oracles A, B et C qui se prononcent sur le fait que deux constantes a et b sont des entiers ou non. On prend en compte l'avis de deux oracles pour décider si une constante est un entier et cette décision ne peut intervenir que s'il y a consensus. Si les oracles ne s'accordent pas, ils doivent se reprononcer. Les règles de cette sorte de «vote» sont les suivantes

- $A(T) \wedge B(T) \rightarrow Entier(a)$

  | *Si A et B répondent vrai en même temps, alors, sans aucun doute, a est un entier.*

- $A(T) \wedge C(T) \rightarrow \neg Entier(b)$

  | *Si A et C répondent vrai en même temps, alors, sans aucun doute, b n'est pas un entier.*

- $Entier(X) \wedge Successeur(X, XX) \rightarrow Entier(XX)$

  | *Si X est un entier et XX est son successeur, XX est aussi un entier.*

Le graphe instancié de la base est le suivant :

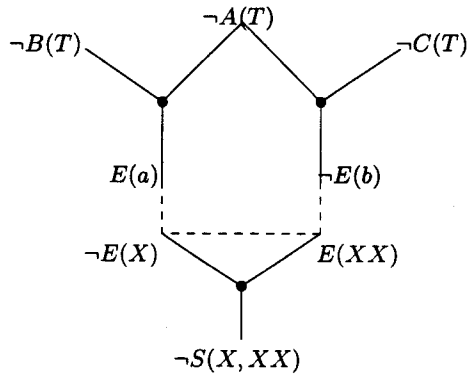


Figure 68: Graphe instancié de la base

On suppose que les faits $S(a, c_1), S(c_1, c_2), \ldots, S(c_n, b)$ sont insérés dans la base de faits, de sorte que $E(a) \rightarrow E(b)$.

Si $B(t)$ et $C(t)$ sont ajoutés aux faits, on peut alors déduire $\neg A(t)$. En effet, si $A(t)$ était vrai, on saurait déduire que a est un entier alors que b ne l'est pas ce qui contredit $E(a) \rightarrow E(b)$

En revanche, quand on ajoute $B(t_1)$ et $C(t_2)$ $(t_1 \neq t_2)$ aux faits, on ne peut rien déduire au sujet de A.

On peut tenter d'achever cette base comme précédemment, c'est à dire en supposant que la tête de cycle est fausse. Dans notre cas, la tête de cycle est $\neg A(X)$. La variable X est ici quantifiée universellement. La négation de la tête est donc après skolémisation $A(c)$. On pourrait alors ajouter les clauses suivantes :

$\neg B(X) \vee E(a \mid A(c))$	initiation
$\neg E(X \mid A(c)) \vee \neg S(X, XX) \vee E(XX \mid A(c))$	propagation
$\neg E(b \mid A(c)) \vee \neg C(X) \vee \neg A(X)$	terminaison

On serait alors capable de produire $A(t)$ à partir de $B(t)$ et $C(t)$, mais on produirait également $A(t_2)$ à partir de $B(t_1)$ et $C(t_2)$, ce qui est incorrect. Ce problème est lié à l'introduction de la constante c.

En fait, briser le cycle ne peut se justifier que s'il y a fusion. Or, il n'y a pas fusion entre $A(t_1)$ et $A(t_2)$ et briser le cycle dans ce cas précis est incorrect. En revanche, quand la base de faits contient $B(t)$ et $C(t)$, il y a réellement une fusion et il faut effectivement briser le cycle dans ce cas.

On s'aperçoit donc qu'il faut vérifier lorsqu'on brise le cycle qu'il y aura bien fusion. Pour ce faire, il faut propager le long du cycle une information permettant de s'en assurer. Il suffit pour cela de vérifier que le $A(X)$ présent dans la clause d'initiation et le $A(X)$ présent dans la clause de terminaison seront bien unifiables à la fin du parcours du cycle. L'information à propager le long du cycle est donc la trace des substitutions effectuées au cours de la résolution sur le prédicat $A(X)$. Pour ce faire, au lieu d'introduire une constante de Skolem pour la négation de la tête $\neg A(X)$, on utilisera $\exists Y, A(Y)$. La variable Y pourra alors conserver la trace des substitutions effectuées. On peut également dire que Y est le témoin de ces substitutions. Pour cette raison, on appelle le littéral placé derrière le symbole | le **témoin de propagation**. Dans la clause d'initiation et de terminaison, le témoin de propagation doit avoir exactement les mêmes variables que la tête de cycle pour justement capturer les substitutions qui affectent la tête. En revanche, les variables du témoin dans

les clauses de propagation doivent être différentes des variables de la clause. En effet, la capture des substitutions s'effectue uniquement lors des résolutions avec les pivots.

Cela conduit donc à ajouter les clauses suivantes à la base :

$$\begin{array}{ll} \neg B(X) \vee E(a|A(X)) & \text{initiation} \\ \neg E(X|A(Y)) \vee \neg S(X, XX) \vee E(XX|A(Y)) & \text{propagation} \\ \neg E(b|A(X)) \vee \neg C(X) \vee \neg A(X) & \text{terminaison} \end{array}$$

Cette fois, on sait toujours dériver $\neg A(t)$ de $B(t)$ et $C(t)$, mais on ne dérive rien sur A de $B(t_1)$ et $C(t_2)$. En effet, dans ce cas, la seule instance pertinente de la clause de terminaison est $\neg E(b|A(t_1)) \vee \neg C(t_1) \vee \neg A(t_1)$ et ne peut être utilisée.

On notera bien que $E(X|A(Y))$ est toujours la notation d'un prédicat que l'on pourrait écrire $E_A(X, Y)$. On notera en particulier que $E(X|A)$ est un prédicat d'arité 1, tandis que $E(X|A(Y))$ est d'arité 2.

Par ailleurs, la base résultante est bien équivalente par ses impliqués à la base initiale, puisque les seuls impliqués premiers de la base achevée qui ne font appel qu'au vocabulaire de la base initiale, à savoir $\neg B(Z) \vee \neg C(Z) \vee S(a, X_1) \vee \neg S(X_1, X_2) \vee \ldots \vee S(X_n, b) \vee A(Z)$, sont bien des impliqués premiers de la base initiale et réciproquement.

15.3.4 Exemple d'achèvement (4)

L'exemple ci-dessous illustre ce qui se passe lors de la saturation, c'est à dire lorsqu'une clause avec fusion doit servir dans un autre cycle où le littéral fusionné sert de pivot.

Soit la base

$$\left\{ \begin{array}{l} \neg E \vee A \\ \neg A \vee B \\ \neg B \vee G \vee \neg C \\ C \vee D \vee E(3) \\ \neg E(X) \vee \neg S(X, XX) \vee E(XX) \\ A \vee \neg E(1) \\ \neg D \vee F(c) \\ \neg F(X) \vee \neg T(X, XX) \vee F(XX) \\ \neg F(a) \vee \neg E \end{array} \right.$$

Son graphe instancié est le suivant :

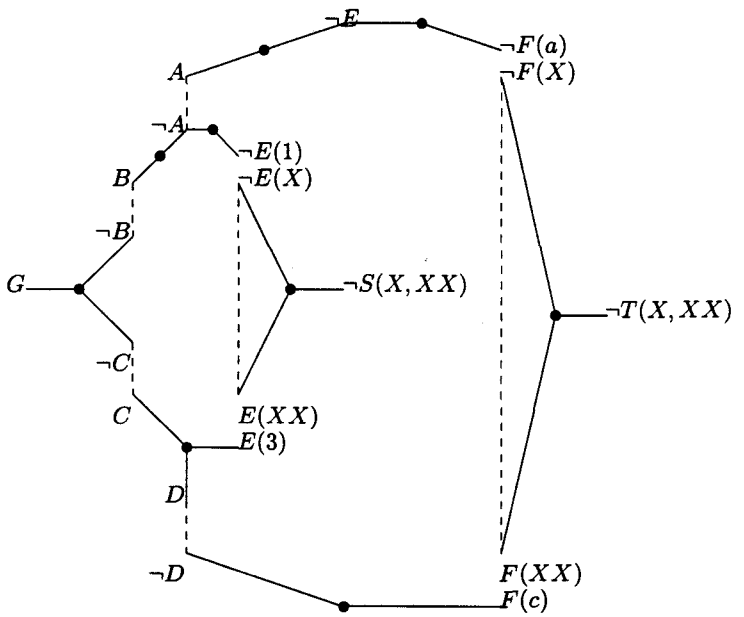


Figure 69: Graphe instancié de la base

Ce graphe comporte plusieurs cycles. Nous nous intéressons à ceux qui vont nous permettre de déduire $\neg E$ à partir des faits $\{\neg G, S(1,2), S(2,3), T(a,b), T(b,c)\}$. On peut d'abord remarquer le cycle dont les piquants sont G et $\neg S(X, XX)$. Ce cycle n'est pas élémentaire. On l'équipe donc comme suit (certaines clauses ont été omises du graphe)

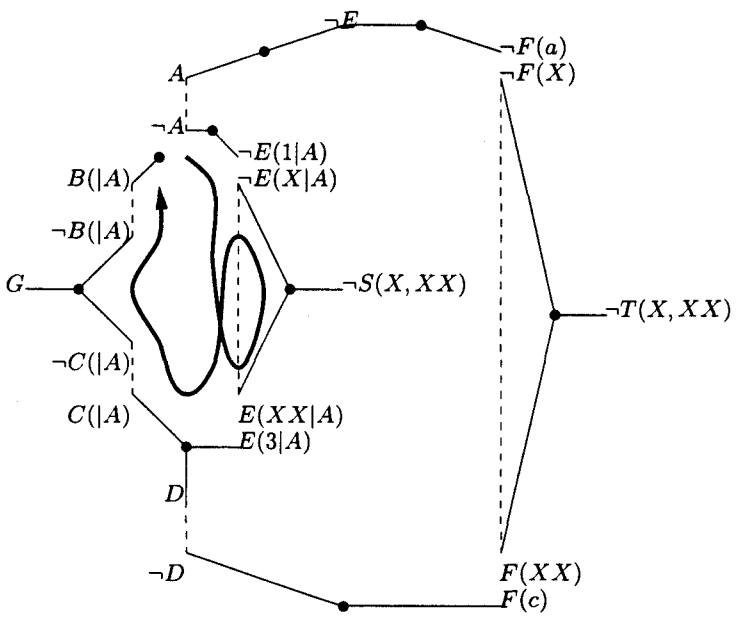


Figure 70: Situation après équipement du premier cycle

À ce moment, il ne subsiste plus qu'un cycle pertinent sur la figure qui n'est pas davantage élémentaire. On l'équipe donc de la même manière lors de la deuxième étape de la saturation. Cela donne ceci.

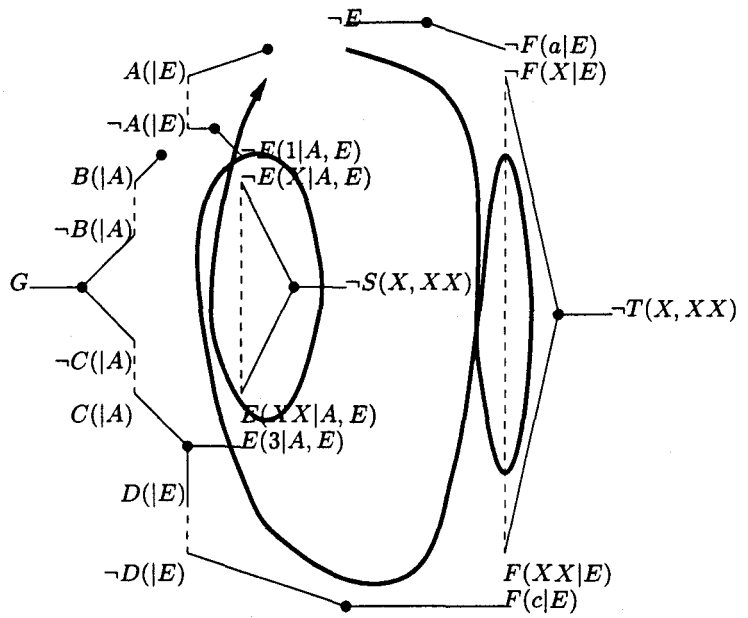


Figure 71: Situation après équipement du second cycle

On est ici amené à avoir plus d'une hypothèse par littéral. Cela n'est en rien gênant. $E(3|A, E)$ représente toujours un littéral qu'on peut noter autrement par $E_{A,E}(3)$. D'autre part, cette accumulation d'hypothèses s'effectue naturellement au cours de la saturation.

La base présentée sur la dernière figure permet bien au chaînage avant de prouver $\neg E$ à partir des faits $\{\neg G, S(1,2), S(2,3), T(a,b), T(b,c)\}$.

15.3.5 Résolutions unitaires préalables

Comme on l'a vu, initier le parcours d'un cycle revient à placer temporairement dans la base de faits la négation de la tête de cycle. Cependant, ce littéral n'est pas réellement placé dans la base de faits. Il n'est donc pas disponible lorsque le chaînage avant en aurait besoin pour effectuer des inférences. Il faut donc prévoir à la compilation les éventuelles résolutions unitaires que la présence de ce littéral dans la base de faits aurait permises. C'est du reste la seule technique employable puisque, le chaînage avant que nous utilisons étant monotone, il est impensable de placer temporairement la tête de cycle dans les faits, voir si le cycle peut être bouclé, pour enfin la retirer de l'ensemble des faits.

Considérons le cas général ci-dessous formé par un cycle avec un prédicat P à son extrémité et contenant une clause avec le prédicat Q.

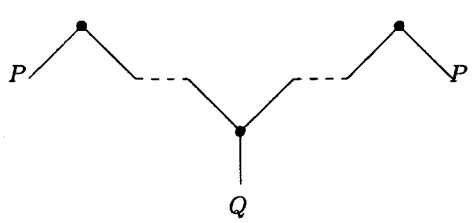


Figure 72: Cas général

On étudie les différents cas possibles d'unification entre P et Q.

– $\exists\sigma, \sigma P = Q$
L'hypothèse émise lors de l'initiation du cycle est plus générale que la condition de propagation. On peut donc effectuer à l'avance l'unit résolution avec l'hypothèse. Dans ce cas, si σ est la substitution qui permet d'unifier à la fois l'hypothèse et la condition de propagation, σ est également appliquée au témoin de propagation.

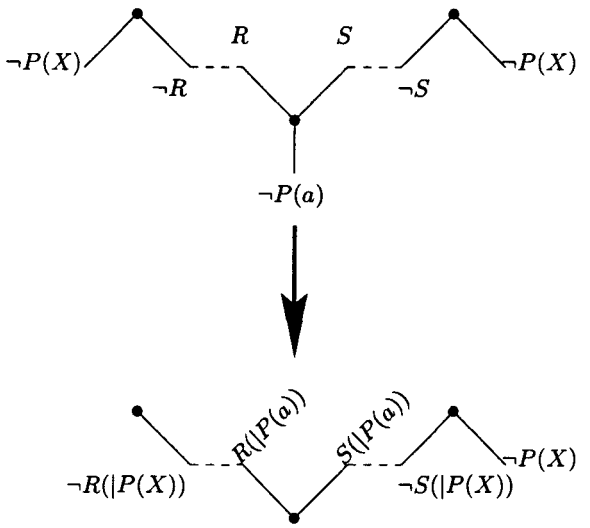


Figure 73: Cas $\exists\sigma, \sigma P = Q$

Exemple 168:
Le chaînage avant sait alors prouver $\neg P(a)$ à partir d'une base de faits vide.

– $\exists\sigma, P = \sigma Q$
L'hypothèse émise lors de l'initiation du cycle est un cas particulier de la condition de propagation. Il faut alors gérer deux cas : celui où la condition de propagation est satisfaite par l'hypothèse d'initiation elle même et celui où la condition de propagation est satisfaite par un autre terme que l'hypothèse.

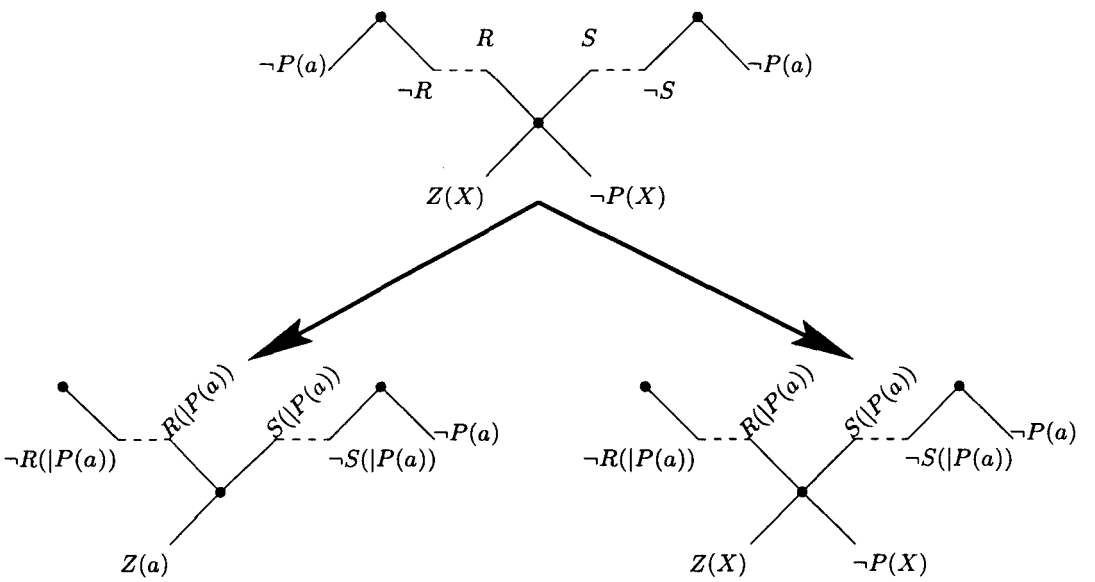


Figure 74: Cas $\exists\sigma, P = \sigma Q$

Exemple 169:

Le chaînage avant sait effectuer l'inférence $\neg Z(a) \rightarrow \neg P(a)$ en utilisant la clause de propagation à gauche sur la figure 74 (cela n'est pas possible avec la clause de droite). Il sait aussi déduire $\neg Z(b) \wedge P(b) \rightarrow \neg P(a)$ en utilisant la clause de droite de la figure 74 alors que cela est impossible sur la clause de gauche.

- $\exists \sigma, \rho, \sigma P = \rho Q$

L'hypothèse émise lors de l'initiation du cycle peut s'unifier avec la condition de propagation. Comme précédemment, il faut gérer deux cas.

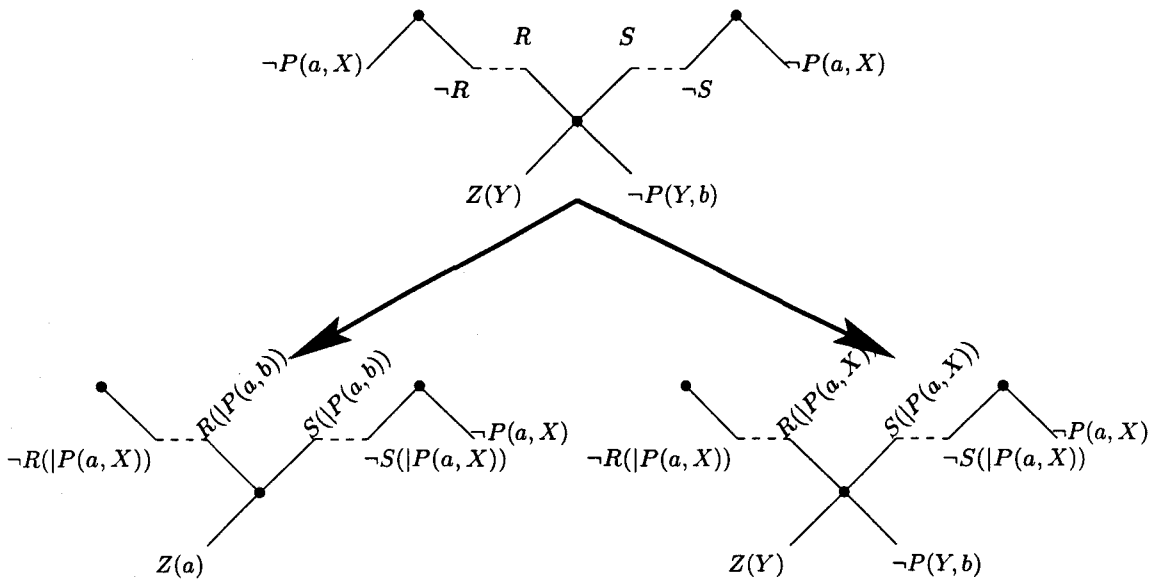


Figure 75: Cas $\exists \sigma, \rho, \sigma P = \rho Q$

Là encore, la substitution qui unifie l'hypothèse avec la condition de propagation doit être appliquée au témoin de propagation lorsqu'on effectue la résolution unitaire préalable.

Exemple 170:

Par chaînage avant, on peut effectuer la déduction $\neg Z(a) \rightarrow \neg P(a, b)$ en utilisant la clause de propagation à la gauche de la figure 75 ce qui est impossible avec la clause de droite. Inversement, la déduction $\neg Z(c) \wedge P(c, b) \rightarrow \neg P(a, X)$ ne peut s'effectuer qu'avec la clause de droite de la même figure.

- $\neg(\exists \sigma, \rho, \sigma P = \rho Q)$

On ne peut unifier la condition de propagation avec l'hypothèse de début de cycle. Il s'agit du cas trivial où l'on ne peut effectuer d'unit-resolution à l'avance.

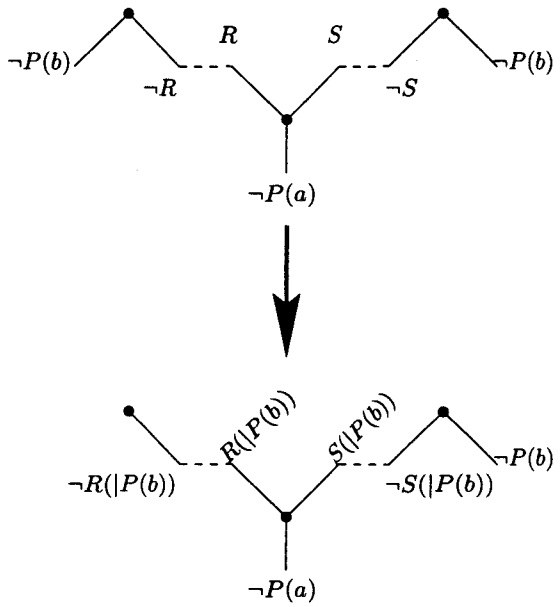


Figure 76: Cas $\neg(\exists\sigma, \rho, \sigma P = \rho Q)$

15.3.6 Indécidabilité de la détection des fusions

Savoir si un cycle correspondra effectivement à une résolution avec fusion n'est pas décidable à l'avance et ce, pour deux raisons. La première est que l'apparition d'une fusion dépend dans certains cas de la base de faits qui sera utilisée par la suite. La seconde est plus profonde et correspond à la notion classique d'indécidabilité : il n'existe pas d'algorithme qui sache dire dans tout les cas s'il y aura fusion ou non.

15.3.6.1 Indécidabilité «faible»

Prenons pour exemple une base qui code les déplacements d'une tortue le long d'une droite graduée.

- $Position(X,T) \wedge Forward(T) \rightarrow Position(X+1, T+1)$

  | *Si la tortue se trouve en X à l'instant T, et qu'elle décide d'aller en avant, alors, elle se retrouve en X+1 à l'instant suivant*

- $Position(X,T) \wedge Backward(T) \rightarrow Position(X-1, T+1)$

  | *Si la tortue se trouve en X a l'instant T, et qu'elle décide d'aller en arrière, alors, elle se retrouve en X-1 à l'instant suivant*

Je souhaite parier avec un ami que la tortue reviendra à son point de départ après un certain temps. Je nomme 0 son point de départ ainsi que son instant de départ. Soit Win la proposition «J'ai gagné mon pari».

J'ajoute donc les deux règles ci-dessous au problème.

- $\neg Position(0,0) \rightarrow Win$

  | *Si la tortue ne part pas de la position 0 à l'instant 0, je considère que la tortue a honteusement triché et je me déclare vainqueur moral.*

- $(\exists T, T > 0, Position(0,T)) \rightarrow Win$

> *Si la tortue se retrouve à la position de départ après son départ, je gagne le pari*

Cette dernière règle est bien une clause quantifiée universellement. La base contient donc les clauses suivantes :

$$\neg Position(X,T) \vee \neg Forward(T) \vee Position(X+1,T+1)$$
$$\neg Position(X,T) \vee \neg Backward(T) \vee Position(X-1,T+1)$$
$$Win \vee Position(0,0)$$
$$Win \vee (T \leq 0) \vee \neg Position(0,T)$$

Le graphe qui lui correspond est représenté ci-dessous, avec un exemple de cycle.

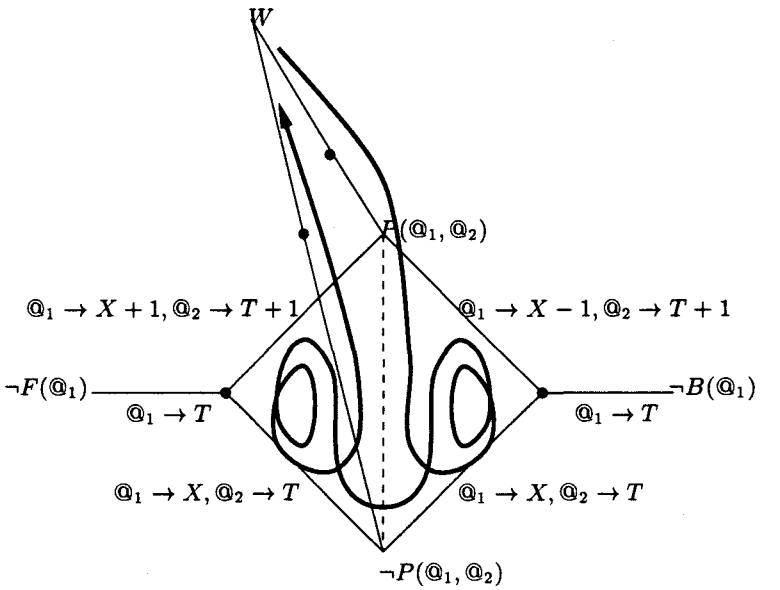


Figure 77: Graphe de la base avec un exemple de cycle

Le dépliage de ce cycle donne ceci :

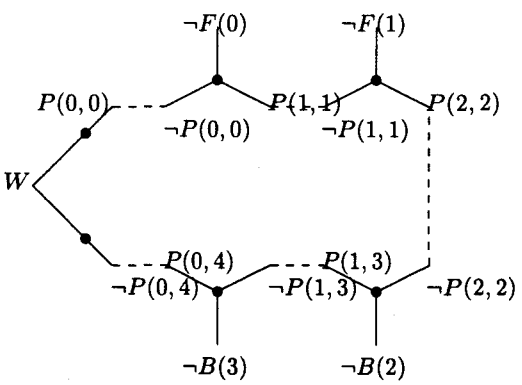


Figure 78: Dépliage du cycle précédent

Le cycle ne peut être refermé que si le nombre de B(T) correspond au nombre de F(T). Cela est très facile à vérifier et n'a rien d'indécidable au sens informatique usuel. Cependant, comme F(T) et B(T) sont a priori des prédicats extensionnels, savoir s'il y aura effectivement fusion ne peut être décidé au moment de la compilation.

15.3.6.2 Indécidabilité «forte»

L'exemple précédent peut être facilement modifié pour faire apparaître une réelle indécidabilité au sens habituel du terme. Même si la base ne dépend pas de faits extensionnels, il est impossible à un algorithme d'affirmer dans tous les cas s'il y aura fusion ou non.

Pour le montrer, nous nous appuyons sur une machine de Turing, que nous représentons par le terme $Turing(Etat, Ruban, Position, T)$. $Etat$ représente l'état de l'automate de la machine de Turing. $Ruban$ est une liste représentant le contenu du ruban. $Position$ représente la position sur le ruban de la fenêtre de lecture. Enfin, T est l'instant où l'on considère l'état de la machine.

Le fonctionnement de la machine peut être traduit par des règles de la forme

- $Turing(Etat, Ruban, Position, T) \rightarrow Turing(E', R', P', T + 1)$

  | *L'état de la machine à un instant T détermine son état à l'instant suivant*

De manière similaire à l'exemple de la tortue, on souhaite parier que la machine s'arrêtera.

- $\neg Turing(init, [], 0, 0) \rightarrow Win$

  | *Pour éviter toute tricherie, je me déclare vainqueur si les conditions initiales sont suspectes.*

- $(\exists T, T > 0, Turing(halt, R, P, T)) \rightarrow Win$

  | *Si l'état puits de l'automate est atteint, la machine est arrêtée et le pari est gagné.*

On vient donc de construire une base qui comporte une fusion sur Win — et implique ce dernier — si et seulement si la machine de Turing considérée s'arrête. Or, il n'existe pas d'algorithme pouvant prédire dans tous les cas si une machine de Turing s'arrête. De ce fait, savoir si un cycle représente effectivement une fusion est indécidable dans le cas général.

Dans une telle machine, la taille du ruban n'est pas bornée. De ce fait, la taille du terme le représentant n'est pas non plus borné. On pourrait alors penser que le problème n'apparaît qu'au premier ordre avec symbole de fonction, ou éventuellement sans symbole de fonction mais avec une base de faits infinie permettant de coder les entiers.

En fait, il n'en est rien puisque le fait de vouloir fonctionner pour toute base de fait finie revient à pouvoir travailler sur des bases de faits infinies (en effet, l'union de toutes les bases de faits finies est une base de faits infinie). Les problèmes d'indécidabilité lors de la compilation sont donc les mêmes dans les cas datalog ou prolog. Il n'en va pas de même à l'exécution. Sans symbole de fonction, on peut toujours décider si un littéral est conséquence ou non. Cela n'est plus vrai lorsque les symboles de fonctions sont autorisés.

15.3.7 Les cycles ambigus

Compte tenu des définitions que nous avons adoptées pour le principe de résolution au premier ordre et de la distinction très nette entre factorisation et résolution binaire, il n'existe plus de cycle ambigu. En effet, le fait que les clauses soient considérées comme des multi-ensembles de littéraux interdit toute factorisation implicite. Or, les cycles ambigus existaient en calcul propositionnel justement parce que l'on pouvait effectuer des factorisations implicites, qui, selon l'ordre de parcours du cycle, pouvait s'effectuer ou non. Lorsqu'on parcourt un cycle au premier ordre, il n'est pas possible de fusionner implicitement l'extrémité d'un pont avec un littéral de la résolvante. Le pont apparaît donc comme un piquant

très ordinaire. Du reste, l'exemple ci-dessous montre que les cycles avec pont ne peuvent plus être éliminés.

Soit la base

$$\begin{cases} A \vee P(a,X) \vee B \\ P(X,b) \vee \neg B \\ \neg P(X,Y) \vee A \vee Q(X,Y) \end{cases}$$

dont le graphe général est représenté ci-après. Le cycle figuré par la flèche contient un pont.

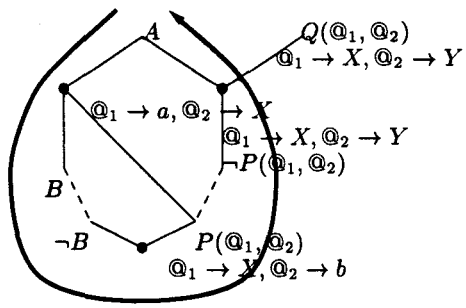


Figure 79: Exemple de cycle au premier ordre avec pont

En suivant ce cycle, on résout d'abord $A \vee P(a,X) \vee B$ et $P(X,b) \vee \neg B$ sur B pour obtenir $A \vee P(a,X) \vee P(Y,b)$. On effectue alors la résolution avec la dernière clause avec $P(Y,b)$ comme pivot (seul choix conforme aux définitions) pour obtenir $A \vee P(a,X) \vee Q(Y,b)$. Cette clause est nécessaire pour pouvoir inférer A à partir de (par exemple) $\neg Q(c,b)$ et $\neg P(a,c)$.

Avec une définition du principe de résolution mélangeant les notions de résolution binaire et de factorisation comme dans [CL73], nous aurions eu trois solutions pour effectuer la résolution entre $A \vee P(a,X) \vee P(Y,b)$ et $\neg P(X,Y) \vee A \vee Q(X,Y)$.

1. résoudre sur le facteur de $P(a,X)$ et $P(Y,b)$ pour obtenir $A \vee Q(a,b)$
   Ceci revient à effectuer une factorisation représentée par un autre cycle.
2. résoudre sur $P(a,X)$ pour obtenir $A \vee P(Y,b) \vee Q(a,X)$
   Ceci revient à suivre un autre chemin dans le graphe que le cycle indiqué.
3. résoudre sur $P(Y,b)$ pour obtenir $A \vee P(a,X) \vee Q(Y,b)$
   C'est la solution décrite par le cycle que nous avons suivi.

Le théorème du calcul propositionnel qui précisait que tout cycle contenant un pont était inutile s'appuyait sur le fait qu'on ne pouvait avoir dans une même clause un prédicat sous forme positive et sous forme négative sans que la clause soit une tautologie. Cela n'est plus vrai au premier ordre (ex : $P(a) \vee \neg P(b)$) et donc les cycles avec pont ne sont plus tous inutiles. Cependant, si le littéral du pont et le littéral de la résolvante avec lequel il fusionne sont exactement égaux, le résultat du calcul propositionnel peut de nouveau s'appliquer et il est inutile de considérer un tel cycle.

15.4 Présentation formelle

Nous récapitulons maintenant formellement la méthode d'achèvement par cycles au premier ordre. Le traitement des cycles élémentaires est similaire au cas propositionnel et

consiste à produire la résolvante linéaire input avec fusion représentée par le cycle

15.4.1 Les cycles élémentaires

Pour chaque cycle élémentaire C de la forme

$$L\text{–}(C_i\text{–}L_i\cdots\neg L_i\text{–})^*C\text{–}L$$

on construit la résolution linéaire input de racine C_0 et de clauses latérales C_1 à C_n n'effectuant que des résolutions binaires et terminé par une factorisation des deux occurrences de la tête de cycle.

Exemple 171:

Soit la base

$$\begin{cases} A \vee P(a,X) \\ \neg A \vee P(X,b) \vee B \\ \neg B \vee P(X,Y) \end{cases}$$

La résolvante linéaire input n'effectuant que des résolutions binaires produit $P(a,W) \vee P(X,b) \vee P(Y,Z)$. On effectue alors la fusion des deux occurrences de la tête de cycle pour obtenir $P(a,X) \vee P(Y,b)$. Les fusions qui peuvent encore être effectuée dans cette clause le seront à l'étape suivante de saturation puisque cette clause forme un cycle à elle seule. La clause produite permet de prouver $P(a,X)$ à partir de $\neg P(c,b)$. Il faut donc bien se contenter de fusionner les deux seules occurrences de la tête de cycle au lieu de tous les prédicats unifiables avec la tête de cycle. En effet, on produirait alors $P(a,b)$ qui ne permettrait pas la déduction précédente.

15.5 Les cycles non élémentaires

Les cycles non élémentaires ne peuvent être ignorés dans le cas général. On notera cependant qu'un cycle qui repasse deux fois par le même point du graphe avec la même instanciation peut être ignoré puisque toutes les instanciations closes de ce cycles sont non élémentaires.

La compilation des cycles non élémentaires nécessite l'introduction de nouveaux symboles de prédicats

Définition 172:

La notation $\boldsymbol{P(X_1,\ldots,X_n|P_1(X_1^1,\ldots,X_n^1),\ldots,P_n(X_1^n,\ldots,X_n^n))}$ représente un nouveau symbole de prédicat qui peut se noter de manière plus conventionnelle

$$P_{P_1,\ldots,P_n}(X_1,\ldots,X_n,X_1^1,\ldots,X_n^1,\ldots,X_1^n,\ldots,X_n^n)$$

Cette notation se lit $P(X_1,\ldots,X_n)$ en supposant $P_1(X_1^1,\ldots,X_n^1)$ à $P_n(X_1^n,\ldots,X_n^n)$

On considère équivalentes les notations $P(X_1,\ldots,X_n)$ et $P(X_1,\ldots,X_n|\varnothing)$ de même que les notations $P(X_1,\ldots,X_n|L\cup L')$ et $(P(X_1,\ldots,X_n|L)|L')$.

On appelle **témoin de propagation** la liste L de prédicats dans l'expression

$$P(X_1, \ldots, X_n|L)$$

Ces nouveaux symboles de prédicats permettent d'obtenir une représentation finie de l'ensemble des impliqués des clauses d'un cycle non élémentaire. On appelle équipement d'un cycle l'opération qui produit cette représentation finie.

Définition 173:

L'**équipement** d'un cycle (élémentaire ou non)

$$L\text{–}C_1\text{–}L_{R_1}\text{---}L_{L_2}\text{–}C_2\text{–}L_{R_2}\text{---}\ldots\ldots\text{---}L_{L_n}\text{–}C_n\text{–}L$$

est l'opération qui consiste à ajouter

- la clause d'**initiation**

$$(C_1 - \{L, L_{R_1}\}) \cup (L_{R_1}|\neg L)$$

- les clauses de **propagation**
  Pour chaque i compris entre 2 et $n-1$, on ajoute la **clause de propagation la plus générale**

  $$P_i = (C_i - \{L_{L_i}, L_{R_i}\}) \cup \{(L_{L_i}|rename(mgl(\neg L))), (L_{R_i}|rename(mgl(\neg L)))$$

  Si l'une des clauses de propagation P a un témoin de propagation L qui peut s'unifier avec un des littéraux $\neg L$ de la clause avec σ comme unificateur, alors, on ajoute une nouvelle clause de propagation qui est la résolvante binaire de P et L avec la substitution σ appliquée au témoin de propagation (**résolution unitaire préalable**).
  L'ensemble des littéraux de P à l'exception de L_{L_i} et L_{R_i} est appelé **condition de propagation**.
- la clause de **terminaison**

$$(C_n - \{L_{L_n}\}) \cup (L_{L_n}|\neg L)$$

Exemple 174:

Soit la base

$$\begin{cases} \neg P(X,Y) \vee \neg R \\ R \vee \neg P(a,X) \vee Z(X) \vee \neg P(Y,b) \vee Z(Y) \vee S \\ \neg P(X,Y) \vee \neg S \end{cases}$$

La clause d'initiation est $\neg R(|P(X,Y))$. La clause de terminaison est $\neg S(|P(X,Y)) \vee \neg P(X,Y)$. La clause de propagation la plus générale est $R(|P(V,W)) \vee \neg P(a,X) \vee Z(X) \vee \neg P(Y,b) \vee Z(Y) \vee S(|P(V,W))$. Elle donne naissance à deux autres clauses de propagation $R(|P(a,X)) \vee Z(X) \vee \neg P(Y,b) \vee Z(Y) \vee S(|P(a,X))$ et $R(|P(Y,b)) \vee \neg P(a,X) \vee Z(X) \vee Z(Y) \vee S(|P(Y,b))$. Chacune de ces clauses produit alors la dernière clause de propagation $R(|P(a,b)) \vee Z(a) \vee Z(b) \vee S(|P(a,b))$.

Proposition 175:

Un cycle équipé est équivalent par ses impliqués au cycle non équipé.

Démonstration 176:

Évident par construction.

15.6 La saturation

Comme en calcul propositionnel, il est nécessaire d'effectuer une saturation selon l'algorithme suivant :

1: $\mathcal{B}_0 \leftarrow \mathcal{B}$
2: $C_0 \leftarrow \mathcal{B}$
3: $i \leftarrow 0$
4:
5: **while** $C_i \neq \varnothing$ **do**
6: $S \leftarrow \varnothing$
7: /* rechercher les cycles élémentaires */
8: Énumérer l'ensemble des cycles élémentaires du graphe général passant par l'une des clauses de C_i et ajouter la clause qu'il représente à S.
9: /* rechercher les cycles non élémentaires */
10: Identifier l'ensemble des cycles non élémentaires du graphe général passant par l'une des clauses de C_i et ajouter leur équipement à S.
11: $C_{i+1} \leftarrow \varnothing$ /* éliminer les tautologies, les clauses w-subsumées et les conséquences qu'on peut obtenir de diverses manières sans que les fusions soient les mêmes */
12: **for all** clause C de S **do**
13: **if** C n'est ni tautologique, ni w-subsumée par une clause de $\mathcal{B}_i \cup C_{i+1}$ **then**
14: **if** $\exists$ au moins une variante de C qui représente une déduction qui ne peut être effectuée par chaînage avant à partir de $\mathcal{B}_i \cup C_{i+1}$ **then**
15: $C_{i+1} \leftarrow C_{i+1} \cup \{C\}$
16: **end if**
17: **end if**
18: **end for**
19:
20: $\mathcal{B}_{i+1} \leftarrow \mathcal{B}_i \cup C_{i+1}$
21: $i \leftarrow i + 1$
22: **end while**

Contrairement au calcul propositionnel, il n'est pas évident que cette saturation termine.

15.7 La saturation termine-t-elle ?

15.7.1 Un premier cas

Soit la base

$$\begin{cases} ①A \vee T(X) \vee Q(X) \\ ②A \vee \neg P(X) \vee Q(s(X)) \\ ③A \vee \neg Q(X) \vee P(t(X)) \end{cases}$$

dont le graphe est le suivant :

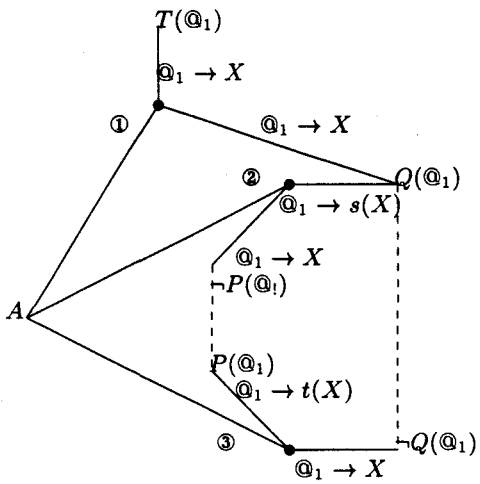


Figure 80: Exemple de saturation infinie

Les clauses ① et ③ forment un cycle qui génère ④$A \vee T(X) \vee P(t(X))$. On peut alors énumérer la succession suivante de cycles

Clauses du cycle	Clauses générées
④, ②	⑤$A \vee T(X) \vee Q(s(t(X)))$
⑤, ③	⑥$A \vee T(X) \vee P(t(s(t(X))))$
⑥, ②	⑦$A \vee T(X) \vee Q(s(t(s(t(X)))))$
⑦, ③	⑧$A \vee T(X) \vee P(t(s(t(s(t(X))))))$
...	

Cette construction est infinie, et aucune des clauses n'est θ-subsumée.

Si l'on associe à chaque clause générée l'ensemble des clauses qui permettent de la produire par résolution, on obtient le tableau ci-dessous

Clauses générées	Clauses ancêtres
⑤$A \vee T(X) \vee Q(s(t(X)))$	①, ③, ②
⑥$A \vee T(X) \vee P(t(s(t(X))))$	①, ③, ②, ③
⑦$A \vee T(X) \vee Q(s(t(s(t(X)))))$	①, ③, ②, ③, ②
⑧$A \vee T(X) \vee P(t(s(t(s(t(X))))))$	①, ③, ②, ③, ②, ③
...	

On s'aperçoit alors que les clauses que l'on génère correspondent à la reconstruction d'un cycle non élémentaire qui sera équipé lors de la première étape de la saturation. Le test de chaînage avant permettra donc d'éviter de produire cette infinité de clauses. Sur cet exemple, la méthode termine.

15.7.2 Un deuxième cas

Dans l'exemple précédent, les littéraux fusionnés étaient toujours les mêmes. L'exemple ci-dessous est construit de manière à éviter ce cas particulier.

Soit la base

$$\begin{cases} ①A \vee B \vee P(X) \\ ②T(X) \vee \neg A \vee P(X) \\ ③\neg P(X) \vee P(s(X)) \vee Q \\ ④\neg A \vee \neg B \vee Q \end{cases}$$

dont voici le graphe général

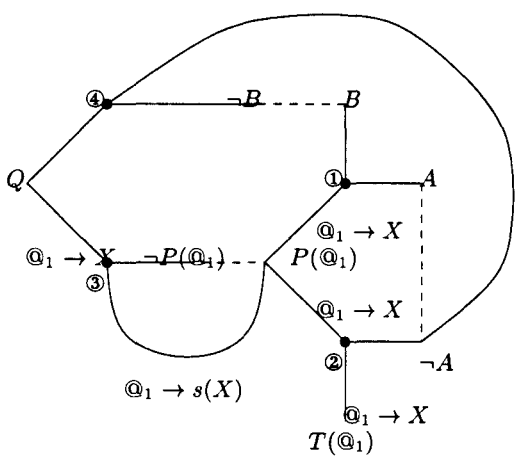


Figure 81: Exemple de saturation infinie

On peut construire à partir de ce graphe une résolution linéaire input infinie comme suit

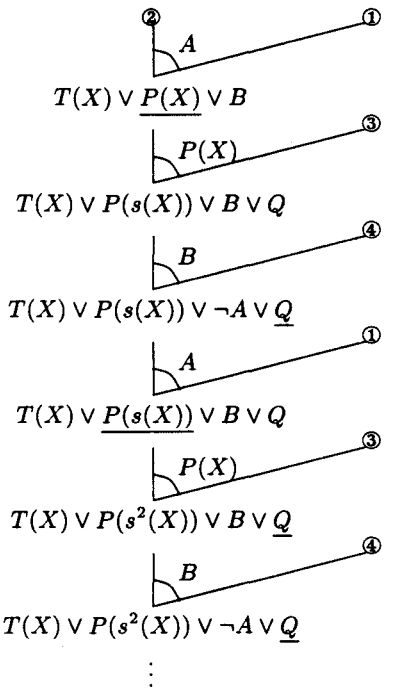


Figure 82: Résolution linéaire infinie

Si l'on essaie de retracer le chemin du graphe (arcs A,B,C et D ci-dessous) qui décrit ces résolutions, on s'aperçoit que ce chemin ne peut en aucun cas correspondre à un cycle représentant une seule résolution linéaire input avec fusion.

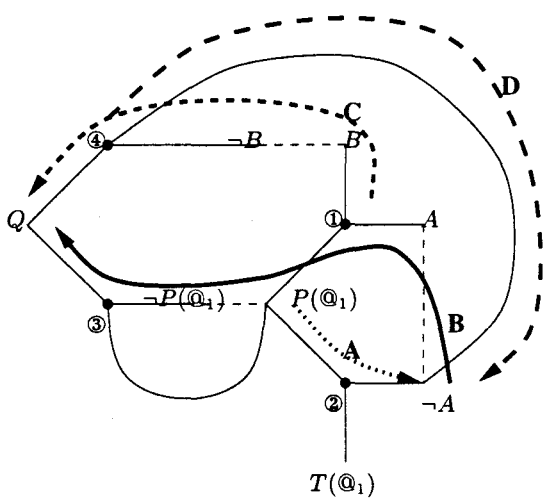


Figure 83: Chemins parcourus sur le graphe

La saturation par recherche de cycles ne terminera donc pas sur cet exemple. Une solution à ce problème est de remarquer qu'il existe des règles simples de transformation de clauses qui permettent de représenter toutes les résolvantes générées. Par exemple, la base ci-dessous est équivalente par ses impliqués à l'ensemble des $T(X) \vee P(s^n(X)) \vee \neg A \vee Q$

$$\begin{cases} T(X) \vee P'(s(X)) \vee \neg A \vee Q \\ \neg P'(X) \vee P'(s(X)) \\ \neg P'(X) \vee P(X) \end{cases}$$

Sur cet exemple, générer une telle représentation est assez facile puisque les clauses utilisées dans la résolution linéaire forment une suite cyclique. Il suffit donc de coder l'effet qu'ont les clauses appartenant à une période. Une fois ce codage effectué, le test par chaînage avant permettra de rendre finie la saturation.

Cependant, ce procédé semble s'apparenter à une démarche inductive et il n'est pas certain qu'on puisse l'appliquer dans tous les cas. On ne sait donc pas à ce jour s'il est possible d'obtenir un algorithme de saturation qui termine dans tous les cas.

15.8 Exemples d'achèvement par cycles

Nous présentons maintenant les résultats obtenus par la méthode d'achèvement par cycles sur les trois bases présentées dans [dV96].

15.8.1 Σ_1

La base Σ_1 contient deux clauses qui codent le calcul de la clôture transitive d'un graphe.

$$\begin{cases} \neg E(X_1, X_2) \vee P(X_1, X_2) \\ \neg E(Y_1, Y_2) \vee \neg P(Y_2, Y_3) \vee P(Y_1, Y_3) \end{cases}$$

Son graphe général est le suivant

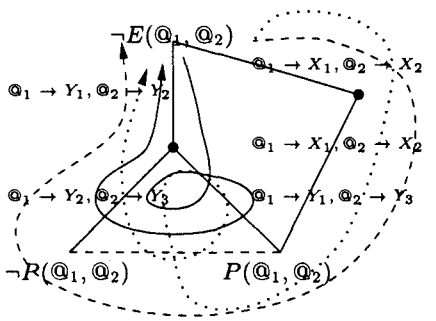


Figure 84: Graphe général de la base Σ_1

Ce graphe contient un cycle élémentaire (ligne en pointillés larges) et deux cycles non élémentaires. Le cycle élémentaire génère $P(X, X) \vee \neg E(X, X)$ qui est w-subsumé par la première clause de la base. Les deux cycles non élémentaires doivent être équipés, ce qui donne :

$P(X_1, X_2 \mid E(X_1, X_2))$	initiation (cycle pointillé)
$P(Y_1, Y_3 \mid E(Y_1, Y_2)) \vee \neg P(Y_2, Y_3)$	initiation (cycle en trait plein)
$\neg P(Y_2, Y_3 \mid E(Y_1, Y_2)) \vee P(Y_1, Y_3 \mid E(Y_1, Y_2))$	propagation (tout cycle)
$\neg P(Y_2, Y_3 \mid E(Y_1, Y_2)) \vee \neg E(Y_1, Y_2) \vee P(Y_1, Y_3)$	terminaison (tout cycle)

L'étape de saturation suivante n'ajoute pas de clause utile à la base. Elle est donc totalement achevée par l'ajout de ces quatre clauses.

15.8.2 Σ_2

La base Σ_2 est une variante de Σ_1

$$\begin{cases} \neg E(X_1, X_2) \vee P(X_1, X_2) \\ \neg E(Y_1, Y_2) \vee \neg P(Y_2, Y_3) \end{cases}$$

Elle contient les mêmes cycles que Σ_1 plus le cycle non élémentaire représenté ci-dessous

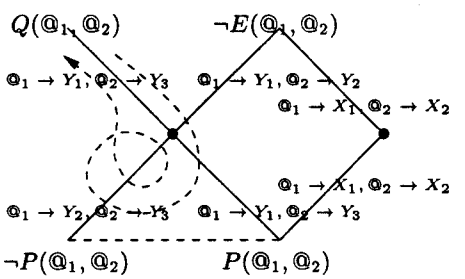


Figure 85: Graphe général de la base Σ_2

L'équipement de ces cycles donne

$P(X_1, X_2 \mid E(X_1, X_2))$	init. (pointillés fin)
$P(Y_1, Y_3 \mid \neg Q(Y_1, Y_3)) \vee \neg E(Y_1, Y_2) \vee \neg P(Y_2, Y_3)$	init. (pointillés larges)
$P(Y_1, Y_3 \mid E(Y_1, Y_2)) \vee \neg P(Y_2, Y_3) \vee Q(Y_1, Y_3)$	init. (trait plein)
$\neg P(Y_2, Y_3 \mid E(Y_1, Y_2)) \vee P(Y_1, Y_3 \mid E(Y_1, Y_2)) \vee Q(Y_1, Y_3)$	prop. (tout cycle)
$\neg P(Y_2, Y_3 \mid E(Y_1, Y_2)) \vee \neg E(Y_1, Y_2) \vee P(Y_1, Y_3) \vee Q(Y_1, Y_3)$	term. (trait plein/point. fin)
$Q(Y_1, Y_3) \vee \neg P(Y_2, Y_3 \mid \neg Q(Y_1, Y_3)) \vee \neg E(Y_1, Y_2) \vee P(Y_1, Y_3)$	term. (pointillé large)

Encore une fois, l'étape suivante de saturation n'apporte rien.

15.8.3 Σ_3

La base Σ_3 s'obtient à partir de Σ_1 en ajoutant deux clauses.

$$\begin{cases} \neg E(X, Y) \vee \neg F(X, Y) \\ E(X, Y) \vee F(X, Y) \\ \neg E(X_1, X_2) \vee P(X_1, X_2) \\ \neg E(Y_1, Y_2) \vee \neg P(Y_2, Y_3) \vee P(Y_1, Y_3) \end{cases}$$

Ces dernières produisent de nouveaux cycles, mais chacun d'entre eux passe deux fois par la même instance de littéral. De ce fait, ils peuvent être ignorés puisque chacune de leurs instanciations closes est non élémentaire. Donc, pour achever Σ_3 il suffit d'ajouter les mêmes clauses que pour la base Σ_1.

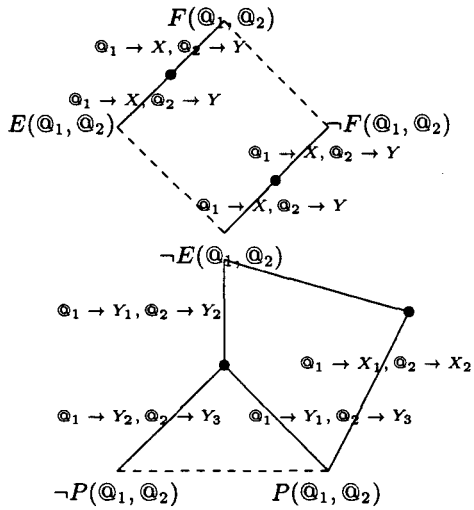


Figure 86: Graphe général de la base Σ_3

[dV96] étudie la compilation approchée de la connaissance en utilisant des bornes inférieures et supérieures de clauses de Horn. Il signale que sa procédure qui améliore celle de [SK91] termine pour Σ_1 et Σ_2, mais pas pour Σ_3. Par opposition, notre procédure termine pour chacune de ces bases et effectue une compilation exacte.

Conclusion

Dans ce mémoire, nous avons présenté de nouveaux résultats sur l'achèvement des bases de connaissances, technique de compilation logique qui permet de rendre complet le chaînage avant quelle que soit la base de faits utilisée par la suite. Nous avons en particulier obtenu en calcul propositionnel un algorithme d'achèvement par parties simple et très efficace qui se fonde sur la notion de graphe. Nous avons également mis en évidence une condition nécessaire et suffisante d'achèvement. Cette dernière a donné jour à une extension de l'achèvement par parties : l'achèvement par cycles. Cette dernière méthode permet d'identifier exactement les clauses qu'il est nécessaire d'ajouter et incorpore de puissants critères d'élagage de l'arbre de recherche. La rentabilité de l'achèvement a alors été expérimentalement démontrée.

Nous avons également étendu la notion d'achèvement aux bases exprimées en calcul des prédicats. En particulier, il a été démontré que l'achèvement fini de ces bases est impossible sans étendre le vocabulaire de la base. Il a également été mis en évidence que la présence ou l'absence de symbole de fonction dans la base à achever changeait très peu les problèmes à surmonter. Trois méthodes d'achèvement ont alors été présentées. La première, l'achèvement par méta-interprète, permet de montrer qu'il est possible d'effectuer un achèvement total au premier ordre mais n'est absolument pas efficace. La deuxième, l'achèvement par champ de production, permet d'effectuer un achèvement partiel de la base sans en étendre le vocabulaire. Les critères définissant cet achèvement sont basés sur des considérations pratiques et font tout l'intérêt de cette méthode. Enfin, la troisième méthode étend l'achèvement par cycles au premier ordre et permet d'obtenir une méthode d'achèvement total assez générale, explicitant l'origine des problèmes de finitude au premier ordre et permettant de les contourner.

Les extensions de ce travail sont nombreuses. Un premier point consiste à optimiser la recherche des cycles. Un deuxième consisterait à identifier les conditions dans lesquelles l'achèvement au premier ordre est rentable. Un troisième consiste à déterminer si l'achèvement par cycles termine dans tous les cas. Enfin, il serait intéressant d'identifier l'usage qui peut être fait de ces techniques dans le domaine des bases de données déductives.

Quatrième partie

Annexes

Annexe A

Bases de test

Les bases propositionnelles utilisées pour tester les performances des algorithmes présentés sont détaillées ci-dessous.

A.1 Les bases de taille fixe

Les bases *députés* et *logiciens* sont tirées de [Car66] et codent des problèmes logiques sur la députation et les logiciens. La base *pannes* est tirée de [Sie87] et code le diagnostic des pannes d'une voiture. Les bases *chandra21, chandra24, easy, ex1, ex2, ex3, ex4, history-ex, selenoid, valve, two-pipes, three-pipes* et *four-pipes* son extraites de [FdK93].

A.2 Les bases paramétrables

Ces bases dépendent d'un ou plusieurs paramètres. On peut donc ajuster leur taille à volonté.

Les bases classiques

Le problème pigeon-n-m consiste à savoir s'il est possible de placer n pigeons dans m boîtes. Seules les instances satisfiables de ce problème nous intéressent ($n < m$). La base ramsey-n encode le problème de Ramsey pour 3 couleurs et n sommets. La base nqueens-n encode le problème des n reines.

L'additionneur

La base adder-n encode l'addition de deux vecteurs de n bits : $a[1..n]$ et $b[1..n]$. Le vecteur $c[0..n]$ est utilisé pour les retenues des additions. Le résultat de l'addition est donné par le vecteur $s[1..n]$ et la dernière retenue $c[n]$.

Le multiplieur

La base mult-n-m encode la multiplication de deux vecteurs de bits $a[1..n]$ et $b[1..m]$. Le résultat de la multiplication est le vecteur $p[1..n+m]$. Les règles encodent la multiplication sous la forme de n additions des produits $a[i].b[1..m]$.

La base mult-inf-n-m contient la base mult-n-m et des règles contraignant le nombre a à être inférieur ou égal à b.

La base mult-sup-n-m contient la base mult-n-m et des règles contraignant le nombre a à être supérieur ou égal à b.

Le démineur

La base $mine - n - m$ encode le problème du jeu du démineur sur un terrain de n par m cases. Les atomes de la base sont

- Mine(x,y)
  vrai s'il y a une mine dans la case de coordonnées (x,y)
- Warning$k$(x,y) pour k de 0 à 8
  vrai s'il y a exactement k mines dans les huit voisines de la case de coordonnées (x,y)

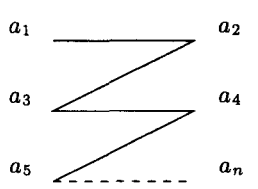
Les règles de la base précisent qu'il n'y a pas de mine sur le bord du terrain et que Warning$k$(x,y) implique qu'il y a exactement k mines dans les huit voisines de la case.

A.3 Les base structurées

Ce sont des bases extraites de [Mat91] et qui ont globalement une structure d'arbre. Elle tentent d'imiter une connaissance hiérarchique, qui est certainement le type de connaissance que l'on retrouve le plus dans les applications des systèmes experts. Leur graphe d'atomes a été représenté à coté de leur définition.

Type1-n

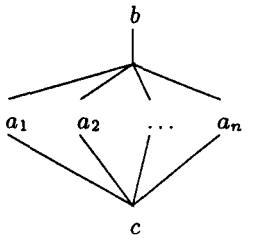
$$\begin{cases} a_1 \rightarrow a_2 \\ \neg a_3 \rightarrow \neg a_2 \\ a_3 \rightarrow a_4 \\ \dots \\ \neg a_{n-1} \rightarrow \neg a_{n-2} \\ a_{n-1} \rightarrow a_n \end{cases}$$



Cette base comporte n atomes et $n-1$ clauses.

Type2-n

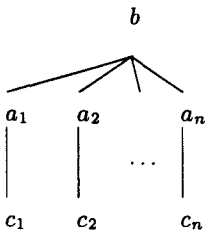
$$\begin{cases} a_1, \ldots, a_n \rightarrow b \\ \neg a_1 \rightarrow c \\ \neg a_2 \rightarrow c \\ \ldots \\ \neg a_n \rightarrow c \end{cases}$$



Cette base contient $n + 2$ atomes et $n + 1$ clauses.

Type3-n

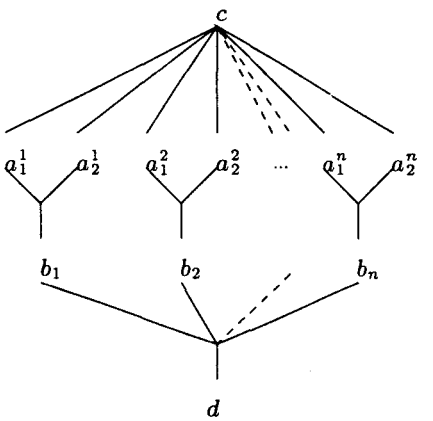
$$\begin{cases} a_1, \ldots, a_n \rightarrow b \\ \neg a_1 \rightarrow c_1 \\ \neg a_2 \rightarrow c_2 \\ \ldots \\ \neg a_n \rightarrow c_n \end{cases}$$



Cette base comporte $2n + 1$ atomes, $n + 1$ clauses.

Type4-n

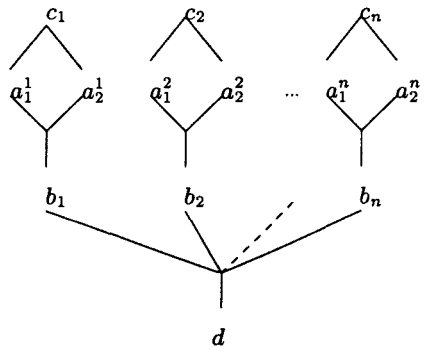
$$i = 1..n \begin{cases} a_1^i, a_2^i \rightarrow b^i \\ \neg a_1^i \rightarrow c \\ \neg a_2^i \rightarrow c \end{cases}$$
$$b^1, \ldots, b^n \rightarrow d$$



Cette base contient $3n + 2$ atomes et $3n + 1$ clauses.

Type5-n

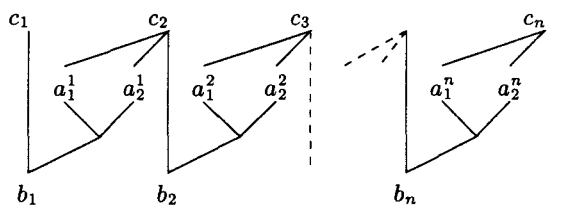
$$i = 1..n \begin{cases} a_1^i, a_2^i \rightarrow b^i \\ \neg a_1^i \rightarrow c^i \\ \neg a_2^i \rightarrow c^i \end{cases}$$
$$b^1, \ldots, b^n \rightarrow d$$



Cette base comporte $4n + 1$ atomes et $3n + 1$ clauses.

Type6-n

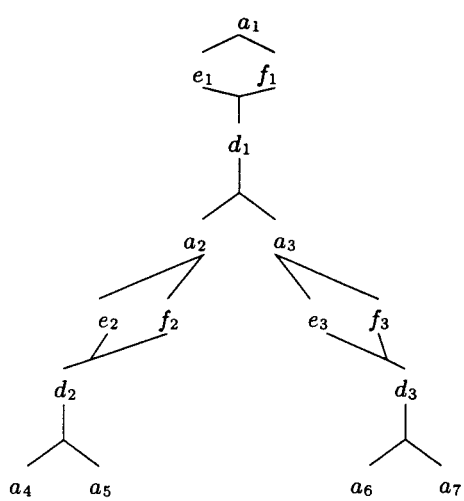
$$i = 1..n \begin{cases} a_1^i, a_2^i \rightarrow b^i \\ \neg a_1^i \rightarrow c^{i+1} \\ \neg a_2^i \rightarrow c^{i+1} \\ b^i \rightarrow \neg c^i \end{cases}$$



Cette base comporte $4n + 1$ atomes et $4n$ clauses.

Type7-n

$$i = 1..n \begin{cases} a_{2i}, a_{2i+1} \rightarrow d_i \\ \neg e_i, \neg f_i \rightarrow \neg d_i \\ e_i \rightarrow a_i \\ f_i \rightarrow a_i \end{cases}$$



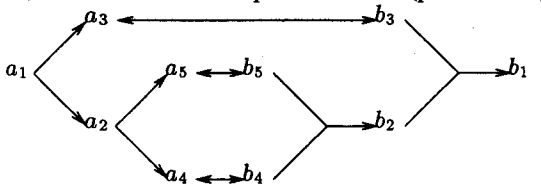
Cette base contient $5n$ atomes et $4n$ clauses.

A.4 Les base cycliques

Ces bases sont construites pour contenir de nombreux cycles (d'où leur nom).

2tree-n

Cette base se représente graphiquement par deux arbres accolés par leurs feuilles via des relations d'équivalence, comme dans l'exemple ci-dessous (pour $n = 5$) :



Le paramètre n représente le nombre d'atomes a. Il doit donc être impair (chaque a a un frère sauf la racine de l'arbre). Le nombre total d'atomes de la base est $2n$ (par symétrie entre les a et b).

Plus formellement, cette base contient les clauses

$$\left.\begin{array}{r} a_i \rightarrow a_{2i} \\ a_i \rightarrow a_{2i+1} \\ b_i \leftarrow b_{2i} \wedge b_{2i+1} \end{array}\right\} \text{ pour } i \text{ de } 1 \text{ à } (n-1)/2$$

$$\left.\begin{array}{r} a_i \rightarrow b_i \\ a_i \leftarrow b_i \end{array}\right\} \text{ pour } i \text{ de } (n+1)/2 \text{ à } n$$

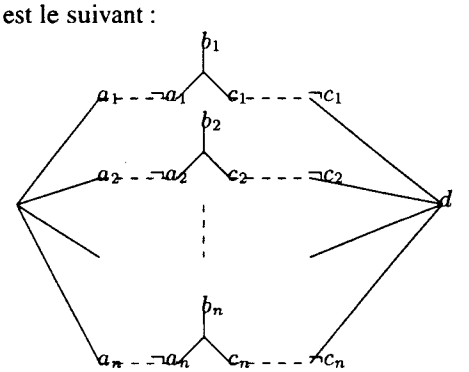
Il y a donc $5n - 1$ clauses.

cycle1-n

Cette base contient les clauses

$$\left.\begin{array}{l} (1)\ \vee_{i \in \{1..n\}} a_i \\ (2) \neg a_i \vee b_i \vee c_i \\ (3) \neg c_i \vee d \end{array}\right\} i \in \{1..n\}$$

Son graphe de littéraux est le suivant :

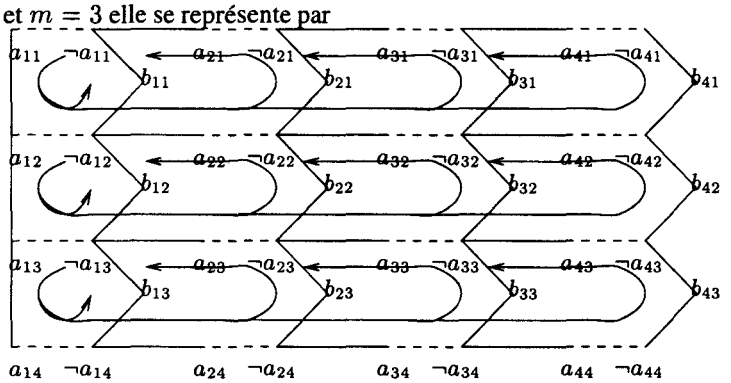


cycle2-n-m

Cette base contient les clauses

$$\left.\begin{array}{l} a_{1,j} \vee a_{1,j+1} \\ \left.\begin{array}{l} \neg a_{i,j} \vee b_{i,j} \\ \neg a_{i,j+1} \vee b_{i,j} \end{array}\right\} i \in \{1..n\} \\ \neg a_{i,j} \vee a_{i+1,j} \quad \} i \in \{1..n-1\}, j \in \{1..m\} \end{array}\right\} j \in \{1..m-1\}$$

Pour $n = 4$ et $m = 3$ elle se représente par

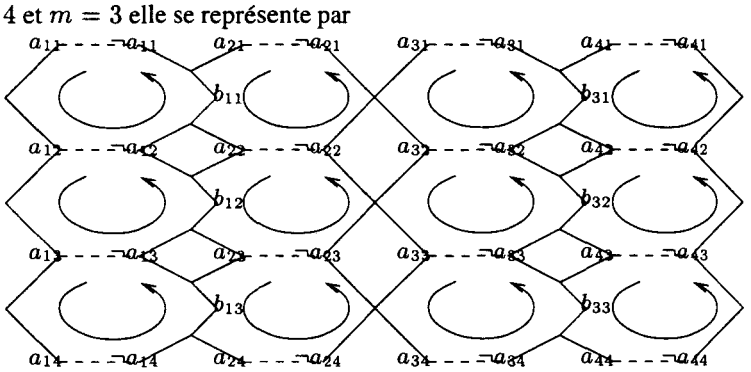


cycle3-n-m

On choisit n pair. Cette base est alors définie par

$$\left.\begin{array}{l} a_{1,j} \vee a_{1,j+1} \\ a_{n,j} \vee a_{n,j+1} \\ \left.\begin{array}{l} \neg a_{i,j} \vee a_{i+1,j} \vee b_{i,j} \\ \neg a_{i,j+1} \vee a_{i+1,j+1} \vee b_{i,j} \end{array}\right\} i \in \{1..n-1\}, i \text{ impair} \\ \neg a_{i,j} \vee a_{i+1,j} \vee \neg a_{i,j+1} \vee a_{i+1,j+1} \quad \} i \in \{1..n-1\}, i \text{ pair} \end{array}\right\} j \in \{1..m-1\}$$

Pour $n = 4$ et $m = 3$ elle se représente par

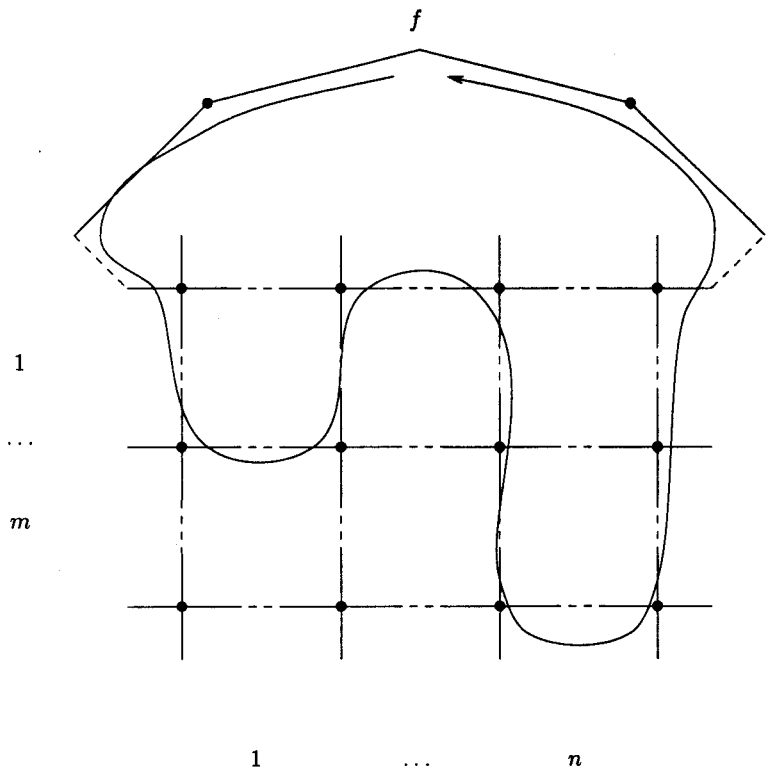


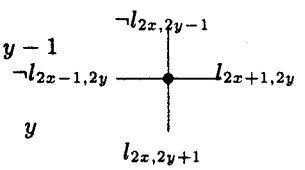
Cycle4-n-m

Cette base est définie par

$$\left\{\begin{array}{l} \left.\begin{array}{l} \neg l_{2x,2y-1} \vee \neg l_{2x-1,2y} \vee l_{2x+1,2y} \vee l_{2x,2y+1} \end{array}\right\} x = 1..n+1, y = 1..m+1 \\ f \vee l_{1,2} \\ f \vee \neg l_{2n+3,2} \end{array}\right.$$

Pour $n = 3$ et $m = 2$, elle se représente par le graphe ci-dessous





Convention de numérotation

Figure 87: Graphe de littéraux de la base cycle4-3-2

Bibliographie

[And68] Peter B. Andrews, *Resolution With Merging*, Journal of the ACM **15** (1968), no. 3, 367–381.

[APT79] B. Aspvall, M. Plass, and R. Tarjan, *A Linear Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulae*, Information Processing Letter **8** (1979), no. 3, 121–123.

[Ber73] C. Berge, *Graphes et hypergraphes*, Dunod, Paris, 1973.

[Bib81] Wolfgang Bibel, *On Matrices with Connections*, Journal of the ACM **28** (1981), no. 4, 633–645.

[BR91] Catriel Beeri and Raghu Ramakrishnan, *On the Power of Magic*, Journal of Logic Programming **10** (1991), 255–299.

[Bry86] Randal E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers **C-35** (1986), no. 8, 677–691.

[Car66] Lewis Caroll, *La logique sans peine*, Hermann, Paris, 1966.

[CC96a] Thierry Castell and Michel Cayrol, *Computation of prime implicates and prime implicants by the Davis and Putnam procedure*, Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction (Budapest), 1996, pp. 61–64.

[CC96b] Thierry Castell and Michel Cayrol, *Une nouvelle méthode de calcul des impliquants et des impliqués premiers*, actes de la 2ème conférence nationale sur la résolution pratique de problèmes NP-complets, CNPC'96 (Dijon), Teknea, Toulouse, mars 1996, pp. 153–167.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee, *Symbolic logic and mechanical theorem proving*, Academic Press, 1973.

[CM94] Olivier Coudert and Jean-Christophe Madre, *Une approche intentionnelle du calcul des implicants premiers et essentiels des fonctions booléennes*, Informatique Théorique et Applications **28** (1994), no. 2, 125–149.

[CS92] M. Cadoli and M. Schaerf, *Approximation in Concept Description Languages*, Principles of Knowledge Representation and Reasoning : Proceedings of the Third International Conference (KR'92) (Cambridge, Massachusetts) (B. Nebel, C. Rich, and W. Swartout, eds.), Morgan Kaufmann, 1992, pp. 330–341.

[DABC94] O. Dubois, P. André, Y. Boufkhad, and J. Carlier, *SAT versus UNSAT*, DIMACS Challenge on Satisfiability Testing, 1994.

[Dal92a] Mukesh Dalal, *Efficient Propositional Constraint Propagation*, Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92) (San Jose, California), July 1992, pp. 409–414.

[Dal92b] Mukesh Dalal, *Tractable Deduction in Knowledge Representation Systems*, Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92) (Boston MA), 1992, pp. 393–402.

[Dal95] Mukesh Dalal, *Tractable Reasoning In Knowledge Representation Systems*, Ph.D. thesis, Graduate School, New Brunswick, New Jersey, May 1995.

[Dec89] Rina Dechter, *Enhancement Schemes for Constraint Processing : Backjumping, Learning,*, Artificial Intelligence **41** (1989), 273–312.

[Del87] Jean-Paul Delahaye, *Forward chaining and computation of two-valued and three-valued models*, Proceedings of the 7th International Conference on Expert Systems and Applications (Avignon), 1987, pp. 1341–1360.

[DLL62] M. Davis, G. Logemann, and D. Loveland, *A Machine Program for Theorem Proving*, Communication of the ACM **5** (1962), 394–397.

[DLP+96] Philippe Devienne, Patrick Lebègue, A. Parrain, Jean-Christophe Routier, and Jörg Würtz, *Smallest Horn Clause Programs*, Journal of Logic Programming **27** (1996), no. 3, 227–267.

[DM93] Jean-Paul Delahaye and Philippe Mathieu, *An Achievement by Part Method to Solve the Incompleteness of Forward Chaining*, actes des IIèmes Journées Francophones de Programmation en Logique, JFPL'93 (Nîmes), Teknea, Toulouse, France, 1993, pp. 155–171.

[Dog95] U. Dogrusöz, *Cyclic structure and colouring of graphs and their parallel solutions*, Ph.D. thesis, Graduate Faculty of Rensselaer Polytechnic Institute, Troy, New-York, July 1995.

[DP60] M. Davis and H. Putnam, *A Computing Procedure for Quantification Theory*, Journal of the ACM **7** (1960), 201–215.

[DP89] Rina Dechter and Judea Pearl, *Tree Clustering for Constraint Networks*, Artificial Intelligence **38** (1989), 353–366.

[DPK82] Narsingh Deo, G.M. Prabhu, and M.S. Krishnamoorthy, *Algorithms for Generating Fundamental Cycles in a Graph*, ACM Transactions on Mathematical Software **8** (1982), no. 1, 26–42.

[DR94] R. Dechter and I. Rish, *Directional Resolution : The Davis-Putnam Procedure Revisited*, Proceedings of the Fourth Conference on Principles of Knowledge Representation (KR'94), 1994, pp. 134–145.

[dV94] Alvaro del Val, *Tractable databases : How to make propositional unit resolution complete through compilation*, KR'94, Proceedings of Fourth International Conference on Principles of Knowledge Representation and Reasoning (J. Doyle, E. Sandewall, and P. Torassi, eds.), 1994, pp. 551–561.

[dV96] Alvaro del Val, *Approximate knowledge compilation : The first order case*, AAAI'96, Proceedings of the Thirteenth National American Conference on Artificial Intelligence, 1996, To appear.

[EBBK89] David W. Etherington, Alex Borgida, Ronald J. Brachman, and Henry Kautz, *Vivid Knowledge and Tractable Reasoning : Preliminary Report*, Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89) (Detroit, Mich.), Morgan-Kaufmann, 1989, pp. 1146–1152.

[EFD95] Yousri El Fattah and Rina Dechter, *Diagnosing tree-decomposable circuits*, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95) (Montreal), 1995, pp. 1742–1749.

[EIS76] S. Even, A. Itai, and A. Shamir, *On the Complexity of Timetable and Multicommodity Flow Problems*, SIAM Journal of Computing **5** (1976), 691–703.

[Esh93] Kave Eshghi, *A Tractable Class of Abduction Problems*, Proceedings of the 30th International Joint Conference of Artificial Intelligence, 1993, pp. 3–8.

[FdK93] Kenneth D. Forbus and Johan de Kleer, *Building Problem Solvers*, MIT Press, 1993.

[GC67] C.C. Gotlieb and D.G. Corneil, *Algorithmes for Finding a Fundamental Set of Cycles for an Undirected Linear Graph*, Communications of the ACM **10** (1967), no. 12, 780–783.

[GEI91] M. Ghallab and E. Escalada-Imaz, *A linear control algorithm for a class of rule-based systems*, Journal of Logic Programming **11** (1991), 117–132.

[Gib69] Norman E. Gibbs, *A Cycle Generation Algorithm for Finite Undirected Linear Graphs*, Journal of the ACM **16** (1969), no. 4, 561–568.

[GM90] Michel Gondran and Michel Minoux, *Graphes et algorithmes*, Eyrolles, 1990.

[Gén96] Richard Génisson, *A propos d'énumération et de polynomialité sur le problème SAT et les Problèmes de Satisfaction de Contraintes*, Ph.D. thesis, Université de Provence, Marseille, Janvier 1996.

[Gré90] Eric Grégoire, *Logiques non monotones et intelligence artificielle*, Hermès, Paris, 1990.

[Héb94] J.J. Hébrard, *A linear algorithm for renaming a set of clauses as a Horn set*, Theoretical Computer Science **124** (1994), 343–350.

[Héb95] J.J. Hébrard, *Unique Horn renaming and Unique 2-Satisfiability*, Information Processing Letters **54** (1995), 235–239.

[Hor87] J. D. Horton, *A Polynomial-Time Algorithm to Find the Shortest Cycle Basis of a Graph*, SIAM Journal on Computing **16** (1987), no. 2, 358–365.

[HR71] Richard C. Holt and Edward M. Reingold, *On the Time Required to Detect Cycles and Connectivity in Graphs*, Mathematical Systems Theory **6** (1971), no. 2, 103–106.

[HT73] John Hopcroft and Robert Tarjan, *Efficient Algorithms for Graph Manipulation*, Communication of the ACM **16** (1973), no. 6, 372–378.

[HW74] L. Henschen and L. Wos, *Unit refutations and Horn sets*, Journal of the ACM **21** (1974), no. 4, 590–605.

[IK77] Tetsuro Ito and Makoto Kizawa, *The matrix rearrangement procedure for graph-theoretical algorithms and its application to the generation of fundamental cycles*, ACM Transactions on Mathematical Software **3** (September 1977), no. 3, 227–231.

[Ino92] Katsumi Inoue, *Studies on Abductive and Nonmonotonic Reasoning*, Ph.D. thesis, Faculty of Engineering of Kyoto University, October 1992.

[Joh75] Donald B. Johnson, *Finding All The Elementary Circuits of a Directed Graph*, SIAM Journal of Computing **4** (1975), no. 1, 77–84.

[Kow75] Robert Kowalski, *A Proof Procedure Using Connection Graphs*, Journal of the Association for Computing Machinery **22** (1975), no. 4, 572–595.

[KS92] Henry Kautz and Bart Selman, *Forming concepts for fast inference*, Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92) (San Jose, Calif.), MIT Press, Cambridge, Mass., 1992, pp. 786–793.

[KT90] Alex Kean and George Tsiknis, *An Incremental Method for Generating Prime Implicants/Implicates*, Journal of Symbolic Computation **9** (1990), 185–206.

[Lep91] François Lepage, *Éléments de logique contemporaine*, Dunod, Montréal, 1991.

[Lev86] Hector J. Levesque, *Making Believers out of Computers*, Artificial Intelligence **30** (1986), 81–108.

[Lov78] Donald W. Loveland, *Automated Theorem Proving : A logical Basis*, North Holland Publishing Company, Amsterdam, 1978.

[Mar93] Pierre Marquis, *Skeptical Abduction*, International Journal on Artificial Intelligence Tools **2** (1993), no. 4, 511–540.

[Mar95] Pierre Marquis, *Knowledge Compilation Using Theory Prime Implicates*, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95) (Montreal), 1995, pp. 837–843.

[Mat91] Philippe Mathieu, *L'utilisation de la logique trivaluée dans les systèmes experts*, Ph.D. thesis, Université des Sciences et Technologies de Lille, France, 1991.

[Mat97] Philippe Mathieu, *Raisonnements centralisés et répartis : De la Compilation Logique à la Coopération entre Agents.*, Ph.D. thesis, Université des Sciences et Technologies de Lille, France, Janvier 1997, Thèse d'Habilitation.

[McA90] D. McAllester, *Truth Maintenance*, Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90), 1990, pp. 1109–116.

[MD76] Prabhaker Mateti and Narsingh Deo, *On Algorithms for Enumerating All Circuits of A Graph*, SIAM Journal of Computing **5** (1976), no. 1, 90–99.

[MD90a] Philippe Mathieu and Jean-Paul Delahaye, *For which bases forward chaining is sufficient ?*, Proceedings of Cognitiva'90 (Madrid), 1990, pp. 699–702.

[MD90b] Philippe Mathieu and Jean-Paul Delahaye, *The logical compilation of knowledge bases*, Proceedings of JELIA'90 (Amsterdam), Lecture Notes in Artificial Intelligence, vol. 478, Springer-Verlag, 1990, pp. 386–398.

[MD94a] Philippe Mathieu and Jean-Paul Delahaye, *A Kind of Achievement by Parts Method*, Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning, LPAR'94 (Kiev, Ukraine) (Frank Pfenning, ed.), Lecture Notes in Artificial Intelligence, vol. 822, Springer-Verlag, 1994, pp. 320–332.

[MD94b] Philippe Mathieu and Jean-Paul Delahaye, *A kind of logical compilation for knowledge bases*, Theoretical Computer Science **131** (1994), 197–218.

[MM96] Bertrand Mazure and Pierre Marquis, *Theory Reasoning within Implicant Cover Compilations*, Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction (Budapest), 1996, pp. 65–69.

[MR72] Eliana Minicozzi and Raymond Reiter, *A Note on Linear Resolution Strategies for Consequence Finding*, Artificial Intelligence **3** (1972), 175–180.

[MS96] Pierre Marquis and Samira Sadaoui, *A New Algorithm for Computing Theory Prime Implicates Compilations*, Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96) (Portland (OR)), 1996, pp. 504–509.

[MT93] Y. Moses and M. Tenneholtz, *Off-line reasoning for on-line efficiency*, Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93) (Chambéry, France), 1993, pp. 490–495.

[Nol80] Helga Noll, *A note on resolution : How to get rid of factoring without loosing completeness*, Proceedings of CADE-5 (Les Arcs, France) (W. Bibel and R. Kowalski, eds.), Lecture Notes in Artificial Intelligence, vol. 87, Springer-Verlag, 1980, pp. 250–263.

[Pat69] Keith Paton, *An Algorithm for Finding a Fundamental Set Of Cycles of a Graph*, Communications of the ACM **12** (1969), no. 9, 514–518.

[Pat71] Keith Paton, *An Algorithm for the Blocks and Cutnodes of a Graph*, Communications of the ACM **14** (1971), no. 7, 468–475.

[PVG96] Tai Joon Park and Allen Van Gelder, *Partitioning Methods for Satisfiability Testing on Large Formulas*, Proceedings of CADE-13 (New Brunswick, NJ, USA) (M.A. McRobbie and J.K. Slaney, eds.), Lecture Notes in Artificial Intelligence, vol. 1104, Springer-Verlag, 1996, pp. 748–762.

[Rau89] Antoine Rauzy, *L'Evaluation Sémantique en Calcul Propositionnel*, Ph.D. thesis, Université de Aix-Marseille II, Faculté des Sciences de Luminy, France, 1989.

[Rau91] A. Rauzy, *Knowledge Extraction in Trivalued Propositional Logic*, Proceedings of Symbolic and Quantitative Approaches to Uncertainty (ECSQAU '91) (Berlin, Germany) (Rudolf Krause and Pierre Siegel, eds.), LNCS, vol. 548, Springer, 1991, pp. 287–291.

[Rau95] Antoine Rauzy, *Polynomial restrictions of SAT : What can be done with an efficient implementation of the Davis and Putnam's Procedure ?*, Proceedings of the International Conference on Principle of Constraint Programming (CP'95) (U. Montanari and F. Rossi, eds.), LNCS, vol. 976, Springer Verlag, 1995, pp. 513–532.

[Rei71] Raymond Reiter, *Two Results on Ordering for Resolution with Merging and Linear Format*, Journal of the ACM **18** (1971), no. 4, 630–646.

[RLK86] J. Rohmer, R. Lescoeur, and J.M. Kerisit, *The Alexander Method – A Technique for The Procesing of Recursive Axioms in Deductive Databases*, New Generation Computing **4** (1986), 273–285.

[RM95a] Olivier Roussel and Philippe Mathieu, *Évaluation des méthodes d'achèvement par parties*, Tech. Report IT-272, Laboratoire d'Informatique Fondamentale de Lille, France, 1995.

[RM95b] Olivier Roussel and Philippe Mathieu, *Évaluation des méthodes d'achèvement par parties*, actes des IVèmes Journées Francophones de Programmation en Logique, JFPL'95 (Dijon), Teknea, Toulouse, France, 1995, pp. 175–189.

[RM96a] Olivier Roussel and Philippe Mathieu, *A New Method for Knowledge Compilation : the Achievement by Cycle Search*, Proceedings of CADE-13 (New Brunswick, NJ, USA) (M.A. McRobbie and J.K. Slaney, eds.), Lecture Notes in Artificial Intelligence, vol. 1104, Springer-Verlag, 1996, pp. 493–507.

[RM96b] Olivier Roussel and Philippe Mathieu, *How to Use Cycles for Logical Compilation*, Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction (Budapest), 1996, pp. 53–60.

[RM96c] Olivier Roussel and Philippe Mathieu, *L'achèvement par cycles des bases de connaissances*, Tech. Report IT-287, Laboratoire d'Informatique Fondamentale de Lille, France, 1996.

[RM96d] Olivier Roussel and Philippe Mathieu, *Une nouvelle méthode de compilation logique : l'achèvement par cycles*, actes des 5èmes Journées Francophones de Programmation Logique et programmation par Contraintes, JFPLC'96 (Clermont-Ferrand), Hermès, Paris, France, 1996, pp. 271–285.

[Rob65] J. A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM **12** (1965), no. 1, 23–41.

[Rou96] Olivier Roussel, *Une méthode de compilation logique par recherche de cycles*, actes des troisièmes Rencontres nationales des Jeunes Chercheurs en Intelligence Artificielle (RJCIA'96) (Nantes) (T. Schaub, L. Siklóssy, and L. Lamarre, eds.), 1996, pp. 189–196.

[RT75] R.C. Read and R.E. Tarjan, *Bounds on Backtrack Algorithms for Listing Cycles, Paths and Spanning Trees*, Networks **5** (1975), 237–252.

[Rym94] Ron Rymon, *A SE-tree-based Prime Implicant Generation Algorithm*, Annals of Mathematics and Artificial Intelligence, special issue on Model-Based Diagnosis **11** (1994).

[Sch96a] Robert Schrag, *Compilation for Critically Constrained Knowledge Bases*, Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96) (Portland (OR)), 1996, pp. 510–515.

[Sch96b] Robert Carl Schrag, *Search in SAT/CSP : Phase Transitions, Abstraction, and Compilation*, Ph.D. thesis, Faculty of the Graduate School, University of Texas, Austin, December 1996.

[SCL70] James R. Slagle, Chin Liang Chang, and Richard C.T. Lee, *A New Algorithm for Generating Prime Implicants*, IEEE Transactions on Computers **C-19** (1970), no. 4, 304–310.

[Sie87] Pierre Siegel, *Représentation et utilisation de la connaissance en calcul propositionnel*, Ph.D. thesis, Université de Aix-Marseille II, Faculté des Sciences de Luminy, France, 1987, Thèse d'Etat.

[SK91] Bart Selman and Henry Kautz, *Knowledge compilation using Horn approximations*, Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91) (Anaheim, Calif.), MIT Press, Cambridge, Mass., 1991, pp. 904–909.

[SK94] Bart Selman and Henry Kautz, *Knowledge Compilation and Theory Approximation*, Journal of the ACM **43** (1994), no. 2, 193–224.

[Sys81] Maciej M. Syslo, *An Efficient Cycle Vector Space Algorithm for Listing all Cycles of a Planar Graph*, SIAM Journal of Computing **10** (1981), no. 4, 797–808.

[Tie70] James C. Tiernan, *An Efficient Search Algorithm to Find the Elementary Circuits of a Graph*, Communications of the ACM **13** (1970), no. 12, 722–726.

[Tis67] Pierre Tison, *Generalization of Consensus Theory and Application to the Minimization of Boolean Functions*, IEEE transactions on electronic computers **16** (1967), no. 4, 446–456.

[VS88] Chvátal Vašek and Endre Szemerédi, *Many Hard Examples for Resolution*, Journal of the ACM **35** (1988), no. 4, 759–768.

[Wei72] Herbert Weinblatt, *A New Search Algorithm for Finding the simple Cycles of a Finite Directed Graph*, Journal of the ACM **19** (1972), no. 1, 43–56.

[WJ66] John T. Welch Jr., *A Mechanical Analysis of the Cyclic Structure of Undirected Linear Graphs*, Journal of the ACM **13** (1966), no. 2, 205–210.

Index des mots clefs

– U –

– V –

– W –

Index des notations

– Symboles –

Abstract

Forward chaining is an inference algorithm which is quite natural since it is based on modus ponens. Besides, it is very simple and therefore very efficient. It is also a production algorithm which aims at producing every fact which is a consequence of the knowledge. However, one drawback is that it is not complete with respect to the usual boolean logic. Achievement offers an interesting solution to this problem. It consists in adding to the knowledge, during a compilation phase, some of its consequences so that forward chaining become complete. Our dissertation extends previous results obtained on this kind of knowledge compilation by Philippe MATHIEU.

In propositional calculus, we first present a very efficient algorithm which splits the knowledge into several parts which are achieved independently. This achievement by parts results in a substantial speedup of the compilation. Then, we obtain a necessary and sufficient condition of achievement from which a new achievement method is derived: the achievement by cycle search. This method is a refinement of the achievement by parts method and allows a precise identification of the rules which must be added to the knowledge for completeness. Another advantage is that it is an incremental and graphical method. Therefore, it easily explains the user why a rule must be added. At last, we prove experimentally that achievement is a real compilation method since it speeds up inferences.

We further extend the notion of achievement to the predicate calculus. We show that the notions used in propositional calculus must be refined for first order bases, for otherwise the achieved base should be of infinite size in some cases. This problem arises for bases with or without function symbols. It can be avoided by extending the vocabulary of the achieved base and using a weaker but plenty satisfactory notion of equivalence. Then, we prove that a total achievement method can be built from these new notions with the description of an achievement by meta-interpreter. We also define another method which avoids extending the vocabulary but only ensures a partial achievement. At last, the achievement by cycle search is lifted to the first order case to obtain a method for total achievement which clearly explains the problems of finiteness.

Keywords: knowledge compilation, forward chaining, completeness, boolean logic, modus ponens, propositional calculus, predicate calculus

Résumé

Le chaînage avant est un algorithme d'inférence très naturel puisqu'il est fondé sur le modus ponens. Il est par ailleurs très simple et de ce fait très efficace. C'est également un algorithme de production qui vise à déduire tous les faits conséquence de la connaissance. Cependant, il a le défaut de ne pas être complet dans le cadre de la logique booléenne classique. L'achèvement offre une solution originale à ce problème et consiste, lors d'une phase de compilation, à ajouter à la connaissance certaines de ses conséquences afin que le chaînage avant devienne complet. Notre travail consiste à étendre les résultats obtenus par Philippe MATHIEU sur cette forme de compilation logique.

En calcul propositionnel, nous présentons tout d'abord un algorithme très efficace qui permet de segmenter la connaissance et d'achever indépendamment chacune des parties obtenues. Cet achèvement par parties permet une accélération spectaculaire des temps de compilation. Nous présentons ensuite une condition nécessaire et suffisante d'achèvement qui donne naissance à une nouvelle méthode : l'achèvement par cycles. Cette méthode est un raffinement de l'achèvement par parties qui permet d'identifier précisément quelles sont les règles qui doivent être ajoutées à la connaissance. Elle a par ailleurs l'avantage d'être incrémentale et graphique. Elle permet donc d'expliquer simplement à l'utilisateur l'ajout des règles. Nous montrons enfin que l'achèvement est une véritable compilation puisqu'elle permet d'effectuer plus rapidement les inférences.

Nous étendons ensuite la notion d'achèvement au calcul des prédicats. Nous montrons que la notion utilisée en calcul propositionnel ne peut pas s'étendre à toute base du premier ordre puisque la base achevée devrait être infinie. Ce problème se pose aussi bien pour les bases avec symboles de fonctions que sans. Il peut cependant être évité en augmentant le vocabulaire de la base achevée et en utilisant une notion d'équivalence plus faible mais parfaitement satisfaisante. Nous montrons alors avec l'achèvement par méta-interprète que cette nouvelle notion permet effectivement de réaliser des achèvements totaux. Nous définissons également une méthode qui évite cette extension du vocabulaire mais ne garantit qu'un achèvement partiel. Enfin, nous étendons l'achèvement par cycles au premier ordre ce qui permet d'obtenir une méthode d'achèvement total expliquant clairement les problèmes de finitude rencontrés.

Mots-clés: compilation logique, chaînage avant, complétude, logique booléenne, modus ponens, calcul propositionnel, calcul des prédicats