Une structure de données pour représenter de grands ensembles de termes égaux

Application à une méthode générale de simplification d'expressions

Mêton Mêton Atindehou

Juin 2018

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics (ICTEAM)
Université catholique de Louvain (UCL)
Louvain-la-Neuve, Belgique

École Polytechnique d'Abomey-Calavi (EPAC)
École Doctorale des Sciences De l'Ingénieur (ED-SDI)
Université d'Abomey-Calavi (UAC)
Abomey-Calavi, Bénin

Thèse soumise en exécution partielle des exigences du grade de :

- Docteur en Sciences de l'Ingénieur et Technologie de l'UCL,
- Docteur en Sciences de l'Ingénieur de l'UAC.

Membres du jury :

Prof. Yves **Deville** (co-promoteur)	ICTEAM, UCL, Belgique
Prof. Baudouin **Le Charlier** (co-promoteur)	ICTEAM, UCL, Belgique
Prof. Aurélien **Goudjo** (co-promoteur)	FAST, UAC, Bénin
Prof. Antoine **Vianou** (co-promoteur)	ED-SDI, UAC, Bénin
Prof. Charles **Pecheur** (président)	ICTEAM, UCL, Belgique
Prof. Pascal **Fontaine**	LORIA, UL, France

Résumé

Nous introduisons une nouvelle structure de données, appelée collection de structures, conçue dans le but de simplifier efficacement des expressions. Nous décrivons précisément sa sémantique et nous en donnons une implémentation optimale. Nous fournissons une étude détaillée de sa complexité théorique que nous complétons par une étude expérimentale approfondie.

Nous utilisons les collections de structures pour calculer la congruence définie par un ensemble d'équations non closes et un ensemble de générateurs, lorsque cette congruence possède un nombre fini de classes d'équivalence. Ce résultat nous permet de minimiser en temps linéaire les expressions appartenant à de telles théories, par exemple, les expressions booléennes utilisant au plus trois variables propositionnelles distinctes.

Nous étendons cet algorithme pour l'appliquer à des théories plus générales comportant trop de classes d'équivalence pour être représentables, en théorie ou en pratique. Nous obtenons ainsi un algorithme générique de simplification d'expressions dont nous démontrons l'utilité pratique en l'appliquant à la simplification d'expressions booléennes comportant jusqu'à 100.000 symboles et 20 variables propositionnelles.

Notre algorithme de calcul de congruence généralise les algorithmes connus de fermeture congruente utilisés en démonstration automatique de théorème. Nous montrons que notre algorithme, basé sur les collections de structures, est plus simple à comprendre et tout aussi efficace que ces méthodes, pour la résolution d'équations closes entre termes.

Indépendamment de ces développements, nous proposons, dans un chapitre préalable, une synthèse d'un ensemble significatif de méthodes de calcul de la liste des impliquants premiers des formules écrites sous forme normale disjonctive. Nous décrivons ces méthodes dans un cadre unifié et nous démontrons rigoureusement leur correction. Nous les implémentons en utilisant une représentation interne unique et nous comparons expérimentalement leur efficacité selon cette implémentation.

Abstract

We introduce a new data structure, called collection of structures, to handle large –often infinite– sets of terms, in order to efficiently simplify terms or expressions. We precisely define the semantics of collections of structures and we provide an optimal implementation thereof. We provide a careful complexity analysis of the new data structure as well as a detailed experimental evaluation of its efficiency.

We use collections of structures to compute the congruence defined by a set of equations between terms with free variables, and a set of generators, in the case where the number of equivalence classes of the congruence is finite. This allows us to minimize expressions belonging to such a theory in linear time. For instance, the method applies to boolean expressions using no more than three distinct propositional variables.

We modify the previous algorithm to make it able to simplify expressions in the context of more difficult situations where the number of equivalence classes is huge or infinite. We experimentally show that the modified algorithm is able to handle expressions using up to $100,000$ symbols with up to 20 different propositional variables.

The algorithm to compute the congruence generalizes the several existing methods to compute the congruence closure of a relation between ground terms, proposed in the literature, to non ground equations. We show that –when limited to ground equations– our method is simpler to understand and, at the same time, as efficient than any previously proposed one.

This thesis also contains an independent –and preliminary– chapter devoted to the problem of computing the list of all prime implicants of a boolean formula in disjunctive normal form. We provide a survey of a significant number of methods proposed in the literature, in a unified way, using a single notation. We carefully prove the correctness of all methods and we experimentally compare their efficiency on the basis of a common compact internal representation of boolean formulas.

Remerciements

J'aimerais remercier le Tout Puissant de la grâce accordée de voir s'achever ce projet de thèse. Cela n'a pu se faire que par le concours de personnes exceptionnelles.

En tout premier, il y a "mon cher Professeur", Baudouin Le Charlier. Je sais le privilège d'avoir pu être à votre école pendant toutes ces années. Vous avez été bien au-delà du rôle traditionnellement dévolu au promoteur de thèse. Vous m'avez accordé votre bienveillante attention, tout en œuvrant méticuleusement à parfaire ma formation. Vous m'avez pour cela donné bien de votre temps. Je suis persuadé que votre coaching, aussi bien sur le plan technique que personnel, portera ses fruits. Votre étudiant reconnaissant vous dit tout simplement merci.

Je remercie aussi Yves Deville pour son soutien à la fois discret et efficace.
Je remercie également Pascal Fontaine, Charles Pecheur, Antoine Vianou, Aurélien Goudjo et Marc Lobelle pour leurs contributions.

Je remercie de leur soutien indéfectible, mon frère Franck, Sandra, bb-Bi, mes parents, ma belle fratrie et mes chers amis.
Enfin, je dis merci à toutes ces autres voix restées anonymes, en jouant le rôle qu'elles devaient jouer pour l'accomplissement de ce projet.

Atindehou Mêton Mêton
Juin 2018

Table des matières

Introduction

Objectifs de ce travail et évolution de la recherche

Les résultats présentés dans cette thèse sont issus d'une recherche ayant comme point de départ le problème général de la simplification des expressions. Par expressions, nous entendons, par exemple, les expressions arithmétiques ou algébriques, les formules logiques ou booléennes, les expressions régulières ou rationnelles, c'est-à-dire des objets "syntaxiques" transformables mécaniquement en d'autres objets de même sorte selon des lois formelles découlant de leur signification mathématique (leur "sémantique"). Par simplification, nous entendons le fait de trouver une expression équivalente qui soit meilleure que l'expression de départ selon un critère précis et calculable. Pour certaines catégories d'expressions, ce critère peut s'exprimer en termes de mise sous forme canonique, c'est-à-dire une forme particulière d'expression permettant de réduire toute expression de cette catégorie en une expression unique. Dans le cas des formules booléennes, par exemple, on peut citer comme formes canoniques : la forme canonique disjonctive ou la liste des impliquants premiers. On peut alors souvent développer des algorithmes de simplification "compositionnels" qui mettent d'abord les sous-expressions sous forme canonique et qui appliquent aux expressions résultantes des règles systématiques pour calculer la forme canonique de l'expression de départ. Cette approche n'est pas toujours applicable car une notion de forme canonique n'existe pas toujours. C'est, par exemple, le cas pour les expressions régulières (dites aussi rationnelles). Par ailleurs, lorsqu'une forme canonique existe, elle ne correspond pas nécessairement à la notion de simplicité attendue d'un point de vue intuitif ou pratique. Ainsi, pour les formules booléennes, on désire souvent obtenir une formule comportant le moins de symboles élémentaires possible, ce qui est loin d'être le cas pour la forme canonique disjonctive et, parfois, encore moins pour la

liste des impliquants premiers [1]. Dans ce travail, nous choisissons comme critère de simplicité le nombre de symboles utilisés pour constituer l'expression. Cela laisse encore place à beaucoup de variations car on peut choisir, par exemple, de minimiser le nombre de caractères de l'expression, ou le nombre de nœuds de sa représentation sous forme d'arbre, ou favoriser certaines formes d'expressions en accordant des poids différents à différents symboles. Cette sorte de critères a l'avantage d'être facilement et efficacement calculable et, en supposant même qu'on s'intéresse surtout à la lisibilité des expressions d'un point de vue purement intuitif, on peut parier que plus une expression est courte, plus elle "a de chances" d'être lisible ou compréhensible.

Ayant posé le problème de cette façon, notre recherche a consisté à mettre au point une structure de données permettant de réaliser efficacement la recherche d'expressions simples dans l'ensemble énorme d'expressions équivalentes à une expression donnée. Nous l'appelons simplement *collection de structures*. Intuitivement, une collection de structures peut être vue comme un graphe orienté dont les sommets représentent des ensembles de termes égaux au sein d'une théorie donnée. L'opération principale sur les collections de structures consiste à y ajouter une contrainte d'égalité entre deux termes représentés dans la collection. La contrainte est immédiatement "propagée" dans la collection de structures de telle façon que les termes égaux soient représentés par le même sommet du graphe. Nous avons aussi découvert une implémentation efficace des collections de structures qui permet de réaliser optimalement la propagation des contraintes d'égalité ainsi que la détermination des termes de taille minimale figurant dans un ensemble de termes égaux. Muni de cet outil, nous avons réalisé divers algorithmes génériques de minimisation et de simplification d'expressions [2] au sein de théories définissables par un ensemble d'équations (comportant des variables universellement quantifiées). Ces algorithmes nous ont permis d'obtenir des résultats expérimentaux intéressants. Nous nous sommes ensuite attachés à justifier rigoureusement la correction et la portée théorique de ces résultats et nous avons confronté cette étude théorique aux travaux publiés dans la littérature. Parallèlement à ce travail, nous avons réalisé une étude des méthodes de calcul des impliquants premiers des formules booléennes,

1. Nous avons choisi de ne pas inclure de référence explicite à la littérature dans cette introduction pour éviter d'insister arbitrairement sur tels ou tels travaux. Des références spécifiques sont évidemment fournies dans la suite de ce travail.

2. Tout au long de ce travail, nous considérons les mots *expression*, *formule* et *terme* comme synonymes et interchangeables mais nous utilisons plutôt l'un ou l'autre en fonction des habitudes en vigueur dans tel ou tel domaine. Dans certains cas toutefois, comme au chapitre 1, nous leur attribuons des rôles spécifiques.

publiées dans la littérature. Cette dernière étude nous a fourni une base de comparaison pour évaluer l'intérêt de notre approche, appliquée à la simplification des formules booléennes.

Dans les pages qui suivent, nous présentons les résultats théoriques et pratiques de la recherche évoquée au paragraphe précédent. Cependant, nous ne présentons pas ces résultats dans l'ordre chronologique de leur élaboration, vu qu'il est plus simple pour nous de définir d'abord les concepts servant à la justification de notre méthode que de la présenter pour commencer, avec des résultats expérimentaux "motivants", puis de revenir en arrière pour préciser sa portée théorique exacte. Le lecteur plus curieux de la portée pratique de notre travail que de son exactitude logique pourra donc, peut-être, commencer la lecture de ce document en consultant les exemples et les résultats expérimentaux figurant à la fin de celui-ci.

Contributions

Une thèse de doctorat est un document technique dont la portée scientifique ne se laisse pas aisément découvrir même pour un lecteur averti. Il est donc d'usage de simplifier la tâche du lecteur, averti ou non, en lui indiquant à l'avance les contributions essentielles du travail présenté dans le dit document. Nous sacrifions ci-dessous à l'usage en indiquant quelles sont, selon nous, les contributions de notre travail.

- Nous introduisons une nouvelle structure de données permettant de représenter de très grands ensembles de termes logiquement égaux, c'est-à-dire : liés entre eux par une relation de congruence. Au niveau abstrait (conceptuel), cette structure de données comporte deux opérations principales : une opération pour ajouter des termes à l'ensemble des termes et une autre pour ajouter et propager une contrainte d'égalité entre certains termes.
- La notion de congruence que nous utilisons est nouvelle par rapport aux notions habituelles de la littérature : elle est fonctionnellement complète, c'est-à-dire qu'elle ajoute automatiquement, à l'ensemble des termes représentés, les termes équivalents, en vertu de la propriété de congruence, à un terme déjà représenté.
- Nous décrivons une implémentation optimale de notre structure de données, au moyen d'entiers et de tableaux d'entiers. Cette implémentation se démarque des implémentations utilisées classiquement par les algorithmes de fermeture congruente de la littérature. Elle n'utilise pas la structure Union-Find et est indépendante d'une appli-

cation particulière.

- Nous fournissons une étude de complexité très détaillée de notre nouvelle structure de données, indépendamment d'une application particulière. Nous la complétons par une étude expérimentale d'efficacité qui corrobore les résultats de l'étude de complexité.
- Nous proposons un algorithme incrémental très efficace pour calculer la taille des termes minimaux de chaque classe d'équivalence de termes représentés dans la structure de données. Le surcoût de l'utilisation incrémentale de cet algorithme est négligeable par rapport au coût des autres opérations.
- Au moyen de notre nouvelle structure de données, nous construisons un algorithme pour calculer la congruence définie par un ensemble fini d'équations non closes et un ensemble fini de générateurs (constantes). Cet algorithme converge lorsque le nombre de classes d'équivalence de la congruence est fini. Il généralise substantiellement les méthodes de calcul de fermeture congruente de la littérature, lesquelles s'appliquent seulement à des équations entre termes clos. Particularisé à ce problème, notre algorithme est aussi efficace que les méthodes de la littérature.
- En combinant l'algorithme de calcul de congruence et l'algorithme de calcul des tailles minimales, nous obtenons un algorithme de minimisation pour les expressions figurant dans une théorie correspondant à une congruence possédant un nombre fini de classes d'équivalence. Nous montrons expérimentalement son efficacité pour les expressions booléennes utilisant au plus trois variables propositionnelles distinctes.
- Nous adaptons l'algorithme de minimisation au cas de congruences possédant un nombre de classes d'équivalence infini ou trop grand pour être représentables en pratique. Nous obtenons ainsi un algorithme de minimisation approchée, c'est-à-dire de simplification. Nous l'avons appliqué à des formules booléennes comportant jusqu'à 20 variables propositionnelles et 100.000 symboles.
- Parallèlement et indépendamment des contributions précédentes, nous proposons une étude comparée d'un ensemble significatif de méthodes, proposées dans la littérature, pour calculer la liste des impliquants premiers d'une formule booléenne sous forme normale disjonctive. Les méthodes sont comparées dans un cadre unifié utilisant une notation plus simple et systématique qu'il n'est d'usage. Elles sont aussi comparées expérimentalement, en utilisant une représentation interne des formules simple et compacte.

- Nous démontrons rigoureusement et en détails tous les résultats énoncés dans ce travail, y compris la correction de tous les algorithmes proposés. Toutes les preuves données sont originales.

Plan de la thèse

L'exposé des résultats de notre travail est structuré comme suit. Dans le chapitre 1, nous présentons une comparaison théorique et expérimentale d'un certain nombre de méthodes de calcul de la liste des impliquants premiers d'une formule booléenne, écrite sous forme normale disjonctive. Les résultats de ce chapitre et les concepts qui y sont utilisés sont totalement indépendants des notions proposées dans les deux chapitres suivants pour expliquer et justifier notre travail relatif aux collections de structures. Il est donc possible d'omettre tout d'abord ce chapitre et de commencer la lecture de ce travail au chapitre 2. Celui-ci présente la notion de collection de structures et les principaux résultats théoriques et expérimentaux qui la concernent, ainsi qu'une comparaison détaillée de ces résultats avec diverses approches proposées dans la littérature dans le domaine de la démonstration automatique de théorèmes. Finalement, le chapitre 3 de ce document explique comment utiliser les collections de structures pour simplifier (ou, dans les cas les plus faciles, minimiser) les expressions en suivant l'optique expliquée au début de cette introduction. Pour évaluer expérimentalement cette approche, nous l'appliquons au cas des formules booléennes et nous utilisons certaines méthodes vues au chapitre 1 pour affiner la portée des résultats obtenus. Le lecteur curieux ou pressé pourra commencer l'étude de ce document en parcourant d'abord les sections 3.2 et 3.3 de ce chapitre pour avoir une première idée de l'intérêt pratique des résultats obtenus au terme de notre recherche.

Chapitre 1

À propos du calcul des impliquants premiers d'une formule boolénne

1.1 Introduction : objectifs et plan du chapitre

Dans ce chapitre, nous étudions le problème de calculer la liste des impliquants premiers d'une formule booléenne. Ce problème constitue une première étape dans la résolution du problème de la minimisation d'une formule booléenne écrite sous forme normale disjonctive. Ce sujet a été étudié par un très grand nombre d'auteurs mais, à notre connaissance, aucun d'entre eux n'a fourni une comparaison approfondie des principales méthodes proposées, incluant une description abstraite de chaque méthode dans un cadre simple et unifié, une représentation des formules dans un format interne unique, une implémentation des algorithmes selon cette représentation et une comparaison explicative de leurs performances. Notre but n'est pas de prouver la supériorité d'une méthode sur les autres mais plutôt de comprendre les points forts et les faiblesses de chacune d'elles. Nous n'avons pas non plus pour objectif d'étudier la totalité des méthodes existantes mais seulement un sous-ensemble significatif auquel les autres méthodes se rattachent.

Le reste de ce chapitre est structuré comme suit. La section 1.2 présente des définitions concernant les formules booléennes. Elles servent à fixer la terminologie utilisée par la suite. Les différents auteurs utilisent des terminologies parfois fort différentes, mais nous nous tenons à une terminologie unique, pour clarifier la comparaison des méthodes. La section 1.3 fournit une présentation abstraite des différentes méthodes choi-

sies, illustrée d'exemples traités "à la main". Nous avons remarqué que certaines méthodes sont plus faciles à appliquer "à la main" que d'autres, plus complexes mais plus efficacement automatisables. La section 1.4 contient la comparaison proprement dite des méthodes présentées à la section 1.3. Elle classe les méthodes selon différents critères et décrit ensuite la structure de données que nous utilisons pour implémenter les différentes méthodes afin de comparer leur efficacité en tant que méthodes automatisées. Nous y justifions donc le choix de cette représentation par rapport à d'autres modes de représentation. Cette section contient également une étude et une comparaison expérimentale des performances des algorithmes correspondant aux différentes méthodes accompagnée d'estimations théoriques de complexité. Nous comparons l'efficacité des méthodes pour différentes classes de formules et nous montrons que le choix d'une méthode efficace doit tenir compte de la classe de formules considérée. Finalement, la section 1.5 contient une synthèse des principales observations obtenues dans notre étude des méthodes de calcul des impliquants premiers.

1.2 Notations et terminologie

Nous supposons connues les notions de formule booléenne et de calcul booléen (voir, par exemple, [Bro90, GH08, Hun33, HS00, NM02, NM03, Sam63, Tha04, Vid91]). Le but de cette section est donc seulement de fixer les notations et la terminologie que nous utilisons dans ce chapitre.

Nous construisons des *formules booléennes* au moyen de *variables propositionnelles* et de *connecteurs logiques*. Nous représentons les variables propositionnelles par les lettres minuscules $a, b, c, \cdots$. C'est pourquoi nous utilisons aussi, et de préférence, le mot "*lettre*" pour désigner les variables propositionnelles. Nous utilisons seulement les trois connecteurs logiques `ET`, `OU` et `NON`. Une *formule booléenne*, au sens général du terme, est donc soit une lettre ; soit une *conjonction*, composée de deux formules booléennes plus simples et d'un connecteur `ET` ; soit une *disjonction*, composée de deux formules booléennes plus simples et d'un connecteur `OU` ; soit une *négation*, composée d'une formule booléenne et d'un connecteur `NON`. Pour écrire concrètement les formules booléennes, on utilise en outre des parenthèses et/ou des conventions de priorité pour les connecteurs logiques, que l'on écrit selon le cas '.' (`ET`), '+' (`OU`), '!' (`NON`). On note aussi souvent les conjonctions par simple juxtaposition de leurs composantes, si cela n'est pas ambigu, et les négations en surmontant la sous-formule niée par une barre.

Nous ne donnons pas plus de précision sur la syntaxe (façon d'écrire) les

formules booléennes au sens général, parce que nous utiliserons uniquement des formules écrites sous *forme normale disjonctive* et que nous fixons une syntaxe très stricte pour ces formules. Dans la suite de ce chapitre, nous utilisons toujours, sauf mention explicite du contraire, le mot "*formule*" (sans plus de précisions) pour désigner une formule écrite sous forme normale disjonctive.

Nous disons qu'une formule (sous forme normale disjonctive, donc) est une suite de $n \geq 0$ termes, séparés par le symbole $+$. Une formule est donc de la forme $t_1 + \cdots + t_n$, où les t_i sont des termes. Si $n = 0$, la formule est vide. On la représente alors par le symbole 0. Un *terme* est une suite de $m \geq 0$ littéraux, notée $lit_1 \cdots lit_m$ ou, alternativement, $lit_1 . \cdots . lit_m$. Si $m = 0$, le terme est vide et s'écrit au moyen du seul symbole 1. Un *littéral* est soit un littéral positif, c'est-à-dire une lettre l ; soit une littéral négatif, c'est-à-dire une lettre "surmontée d'une barre" $\overline{l}$.

Nous disons qu'une formule *utilise* une lettre, si cette lettre figure dans au moins un littéral (d'au moins un terme) de la formule.

Pour systématiser et simplifier les manipulations des formules, nous introduisons des contraintes supplémentaires dans leur syntaxe. Nous supposons avoir défini un ordre total sur l'ensemble des lettres. En pratique, dans les exemples, nous utilisons l'ordre alphabétique : $a < b < c < \cdots$. Nous imposons aux littéraux des termes de porter sur des lettres distinctes. De plus, les littéraux doivent être rangés dans l'ordre de leurs lettres. Avec ces contraintes, deux termes logiquement équivalents doivent s'écrire exactement de la même façon. Nous n'écrivons pas, par exemple, $\overline{c}ab$ ou $ba\overline{c}$ mais uniquement $ab\overline{c}$. Nous supposons ensuite qu'un ordre total est défini sur les termes et que les termes d'une formule sont rangés selon cet ordre total. (Nous ne précisons pas davantage ici cet ordre total. L'ordre total effectivement choisi dans notre implémentation est décrit à la section 1.4.) Deux formules utilisant le même ensemble de termes s'écrivent donc exactement de la même façon. La formule $F = \overline{a}\overline{b}\overline{c} + a + b\overline{c}$ est écrite en respectant l'ordre utilisé dans notre implémentation mais les formules $a + \overline{a}\overline{b}\overline{c} + b\overline{c}$ et $a + b\overline{c} + \overline{a}\overline{b}\overline{c}$ ne le sont pas.

Nous utilisons aussi les notations suivantes. Soient *lit*, un littéral contenant la lettre l et soit t un terme qui ne contient pas la lettre l. Nous notons *lit.t*, le terme obtenu en ajoutant le littéral *lit*, à sa place, dans t.

Soient F, une formule n'utilisant pas la lettre l et *lit*, un littéral qui l'utilise. Nous notons *lit.F*, la formule obtenue en ajoutant le littéral *lit*, à sa place, dans chaque terme de F.

Soit une formule utilisant m lettres distinctes ($m \geq 0$). Nous disons

que cette formule est écrite sous *forme canonique* ou, simplement, *est canonique*, si tous ses termes contiennent exactement m littéraux. Une formule canonique contient donc, au plus, 2^m termes. Toute formule, écrite sous forme normale disjonctive ou sous forme générale, est logiquement équivalente à une et une seule formule canonique utilisant les mêmes lettres. Nous appelons, de plus, *terme complet* d'une formule tout terme de la formule canonique qui lui est équivalente et qui utilise le même nombre de lettres. La formule canonique logiquement équivalente à F, ci-dessus, est $\overline{a}\overline{b}\overline{c} + a\overline{b}\overline{c} + \overline{a}b\overline{c} + ab\overline{c} + a\overline{b}c + abc$.

Nous appelons *impliquant* d'une formule tout terme qui implique logiquement cette formule. Les impliquants de F sont les termes $\overline{c}$, $\overline{a}\,\overline{c}$, $\overline{b}\overline{c}$, $\overline{a}\overline{b}\overline{c}$, a, $a\overline{b}$, $a\overline{c}$, $a\overline{b}\overline{c}$, $b\overline{c}$, $\overline{a}b\overline{c}$, ab, $ab\overline{c}$, ac, $a\overline{b}c$ et abc. Nous appelons *impliquant premier* d'une formule tout impliquant qui n'implique pas d'autre impliquant de cette formule que lui-même. Les impliquants premiers de F sont les termes $\overline{c}$ et a.

Pour toute formule, il existe une et une seule formule qui lui est logiquement équivalente et dont l'ensemble des termes est l'ensemble des impliquants premiers de la formule de départ. Nous disons, par convention de langage, que cette dernière formule est la *liste des impliquants premiers* de la formule de départ. La liste des impliquants premiers de F est $\overline{c} + a$.

Le sujet de ce chapitre est l'étude des méthodes permettant de calculer (efficacement) la liste des impliquants premiers d'une formule donnée. La plupart de ces méthodes supposent que la formule de départ est donnée sous forme canonique. Le calcul de la forme canonique d'une formule quelconque F est facile à réaliser, mais a une complexité en $O(n \times 2^m)$, où m est le nombre de lettres utilisées par F et n est le nombre de termes de F. Le coût du calcul de la forme canonique doit donc être pris en compte dans l'évaluation de l'efficacité des méthodes.

1.3 Méthodes de calcul des impliquants premiers d'une formule booléenne

1.3.1 Méthode de Karnaugh

Les objectifs de Karnaugh étaient de mettre en œuvre une méthode simple et rapide de construction de circuits logiques minimaux à des fins uniquement "pratiques", pour des personnes qui devaient concevoir ces circuits "à la main".

Présentation simplifiée de la méthode de Karnaugh

La méthode de Karnaugh [Kar53], est une méthode *visuelle* pour simplifier les formules. Elle consiste à construire un tableau à 2 entrées dont chaque case ou cellule correspond à un terme complet de m littéraux, où m est le nombre de lettres distinctes figurant dans la formule à simplifier.

Les colonnes (et les lignes, quand il y a plus de 2 lettres) doivent être ordonnées de telle sorte que les termes qui identifient deux colonnes successives ne diffèrent que par un et un seul littéral. C'est toujours possible de le faire. Un tel ordonnancement correspond au *code de Gray* [Gra53] qui ordonne les nombres représentés en binaire de telle sorte que deux nombres successifs ne diffèrent que d'un bit. Le code de Gray pour les nombres binaires de 2 bits est 00, 01, 11, 10.

On place un 1 dans chaque cellule correspondant à un terme complet de la formule à simplifier. On repère ensuite *visuellement* les blocs rectangulaires comportant un nombre de cellules égal à une puissance de 2.

Un bloc de 2^l cellules correspond à un terme de taille $m - l$, considérablement plus simple que la formule canonique formée des termes correspondant au bloc de cellules considéré. (Cette formule comporte $m \times 2^l$ littéraux au lieu de $m - l$).

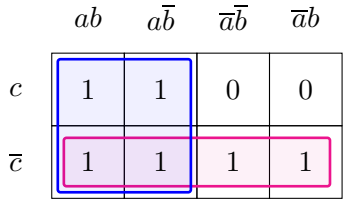
Pour déterminer la liste des impliquants premiers, on choisit *visuellement* les blocs recouvrant toutes les cellules contenant un 1. Il faut s'efforcer de le faire avec des blocs les plus grands possibles. Les impliquants premiers correspondent donc aux blocs maximaux, i.e. ceux qui ne sont pas inclus dans des blocs plus grands (remplis de 1).

Prenons par exemple la formule $F = \overline{a}\overline{b}\overline{c}+a+b\overline{c}$. On prend chaque terme de la formule, on calcule sa suite de termes complets et on remplit les cellules correspondantes. On obtient un terme complet d'un terme t, en lui ajoutant un des deux littéraux de chaque lettre de F qui n'est pas dans t. Les suites de termes complets de $\overline{a}\overline{b}\overline{c}$, a et $b\overline{c}$ sont respectivement $\overline{a}\overline{b}\overline{c}$, $a\overline{b}\overline{c} + ab\overline{c} + a\overline{b}c + abc$ et $\overline{a}b\overline{c} + abc$. On obtient les cellules (voir

table 1.1a) de la table de Karnaugh représentée à la table 1.1.

	ab	$a\overline{b}$	$\overline{a}\overline{b}$	$\overline{a}b$
c	1	1	0	0
$\overline{c}$	1	1	1	1

(a) Cellules de F



(b) Blocs maximaux de F

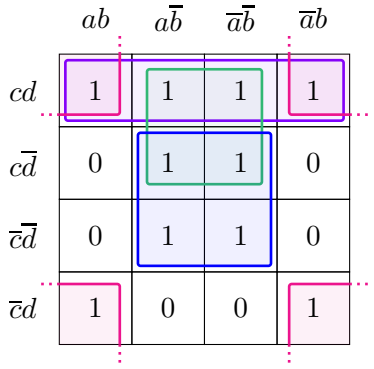
Table 1.1 – Table de Karnaugh de $F = \overline{a}\overline{b}\overline{c} + a + b\overline{c}$

Il y a deux blocs de quatre cellules contenant des 1 (voir table 1.1b). Le premier bloc correspond à la simplification $a\overline{b}\overline{c} + ab\overline{c} + a\overline{b}c + abc = a$, tandis que le second est celui de la simplification $\overline{a}\overline{b}\overline{c}+a\overline{b}\overline{c}+\overline{a}b\overline{c}+ab\overline{c} = \overline{c}$. La liste des impliquants premiers est donc $\overline{c} + a$.

Considérons maintenant la formule $G = \overline{a}\overline{b}\overline{c}\overline{d}+a\overline{b}\overline{c}\overline{d}+\overline{a}\overline{b}c\overline{d}+a\overline{b}c\overline{d}+\overline{a}b\overline{c}d+ab\overline{c}d+\overline{a}\overline{b}cd+a\overline{b}cd+\overline{a}bcd+abcd$. Elle est déjà sous forme canonique. On remplit donc directement les cellules correspondantes à ses termes et on obtient la table de Karnaugh représentée à la table 1.2.

	ab	$a\overline{b}$	$\overline{a}\overline{b}$	$\overline{a}b$
cd	1	1	1	1
$c\overline{d}$	0	1	1	0
$\overline{c}\overline{d}$	0	1	1	0
$\overline{c}d$	1	0	0	1

(a) Cellules de G



(b) Blocs maximaux de G

Table 1.2 – Table de Karnaugh de G

Il y a quatre blocs (voir table 1.2b) de quatre cellules contenant des 1, dont un est plus difficile à visualiser. Il contient les cellules (qui doivent être considérées comme contiguës) correspondant aux termes $\overline{a}b\overline{c}d$, $ab\overline{c}d$, $\overline{a}bcd$ et $abcd$. La liste des impliquants premiers de G est donc $\overline{b}\,\overline{d} + \overline{b}c + bd + cd$.

Limitation de la méthode de Karnaugh

Certains auteurs ont étendu la méthode à plus de quatre variables mais il a été montré dans [Le 14] que ce n'est pas correct car l'aspect visuel de la méthode (repérage de blocs de 1) disparaît. Karnaugh avait étendu sa méthode à six variables [Kar53] mais c'était en utilisant un appareillage à trois dimensions, peu réaliste en pratique.

1.3.2 Méthodes de Quine

Motivations

L'objectif de Quine est de concevoir un procédé "mécanique" utilisable en pratique pour la réduction de formules booléennes écrites sous forme normale disjonctive. Quine tire comme conséquence des travaux de Shannon [Sha38], qu'un procédé du genre permettra également de simplifier les circuits électriques de l'ingénierie.

Première méthode

La première méthode de Quine [Qui52] s'applique uniquement à des formules canoniques. (Mais on peut bien sûr l'utiliser pour des formules quelconques en les mettant d'abord sous forme canonique.) Elle consiste à opérer toutes les fusions applicables aux termes de la formule et à éliminer de celles-ci les termes ayant été fusionnés jusqu'à ce qu'aucune fusion ne soit plus possible.

L'opération de *fusion* s'applique à deux termes distincts ayant le même nombre de littéraux portant sur les mêmes lettres. Si les termes ne diffèrent que par un littéral, ils sont dits *fusionnables* et le résultat de leur fusion est le terme obtenu en retirant des deux termes les littéraux qui diffèrent. Si les termes diffèrent par plus d'un littéral, ils ne sont pas fusionnables.

Quine ne dit pas comment organiser efficacement la recherche des fusions à opérer. On doit donc *a priori* tester toutes les paires de termes portant sur les mêmes lettres.

Nous appliquons cette méthode pour calculer les impliquants premiers de la formule $F = \overline{a}\overline{b}\overline{c}+a\overline{b}\overline{c}+\overline{a}b\overline{c}+a\overline{b}c+\overline{a}bc+abc$. Il y a 6 termes complets et donc 6^2 couples de termes à comparer. Nous présentons les fusions potentielles dans la table 1.3. La cellule se trouvant à l'intersection de la ligne du terme t et de la colonne du terme t' contient le résultat de la fusion de t et t', s'ils sont fusionnables et /, sinon. Tous les termes de F sont fusionnables avec au moins un autre terme. Ce ne sont donc pas des impliquants premiers. On obtient six nouveaux impliquants qui constituent la formule $F_1 = \overline{a}\,\overline{c} + \overline{b}\overline{c} + a\overline{b} + \overline{a}b + ac + bc$. Aucun terme

t/t'	$\overline{a}\overline{b}\overline{c}$	$a\overline{b}\overline{c}$	$\overline{a}b\overline{c}$	$a\overline{b}c$	$\overline{a}bc$	abc
$\overline{a}\overline{b}\overline{c}$	/	$\overline{b}\overline{c}$	$\overline{a}\,\overline{c}$	/	/	/
$a\overline{b}\overline{c}$	$\overline{b}\overline{c}$	/	/	$a\overline{b}$	/	/
$\overline{a}b\overline{c}$	$\overline{a}\,\overline{c}$	/	/	/	$\overline{a}b$	/
$a\overline{b}c$	/	$a\overline{b}$	/	/	/	ac
$\overline{a}bc$	/	/	$\overline{a}b$	/	/	bc
abc	/	/	/	ac	bc	/

Table 1.3 – Fusions des termes de $F = \overline{a}\overline{b}\overline{c} + a\overline{b}\overline{c} + \overline{a}b\overline{c} + a\overline{b}c + \overline{a}bc + abc$

de F_1 n'est fusionnable avec un autre. La formule F_1 constitue donc la liste des impliquants premiers de F.

Seconde méthode

Dans la seconde méthode proposée par Quine [Qui55], on calcule la liste des impliquants premiers à partir d'une formule quelconque (sous forme normale disjonctive) en appliquant deux opérations : l'opération de *consensus*, qui ajoute un nouveau terme à la formule courante, et l'opération de *simplification* qui retire de la formule un terme qui implique logiquement un autre terme de la formule.

L'opération de consensus généralise l'opération de fusion utilisée par la première méthode de Quine. Elle est définie pour deux termes n'ayant qu'un seul littéral différent portant sur la même lettre. Par contre, les autres littéraux peuvent porter sur des lettres différentes ou doivent être égaux s'ils portent sur la même lettre. Le résultat de l'opération est le terme composé de tous ces autres littéraux. Par exemple, le consensus de $\overline{a}bce$ et de $\overline{a}\,\overline{c}df$ est $\overline{a}bdef$ et celui de cdf et de $b\overline{c}$ est bdf.

Quine suggère aussi d'opérer toutes les simplifications applicables avant chaque application de la règle du consensus. Il démontre qu'en procédant de la sorte, on obtient la liste des impliquants premiers lorsque qu'aucune simplification n'est possible et que toute application de la règle du consensus produirait un terme appartenant déjà à la formule courante.

Nous pouvons montrer que ce résultat est correct en raisonnant par induction sur la taille des impliquants de la formule de départ. Soient F une formule et t, un terme. Si F vérifie la condition proposée par Quine et si t est un impliquant de F, alors t implique un terme de F. C'est sûrement vrai si t est un terme utilisant toutes les lettres de F, puisqu'il doit alors figurer dans la forme canonique de F. Sinon, il existe au moins une lettre l de F qui ne figure pas dans t. Les termes $\overline{l}.t$ et $l.t$ sont des impliquants de f et on peut donc supposer, par hypothèse d'induction, qu'ils impliquent chacun un terme de F. S'ils impliquent un terme qui

ne contient pas la lettre l, t implique aussi ce terme et le théorème est prouvé. Sinon $\overline{l}.t$ doit impliquer un terme qui contient le littéral $\overline{l}$ et $l.t$ un terme qui contient le littéral l. L'opération de consensus est applicable à ces deux termes et produit un terme impliqué par t. Puisque F vérifie la condition de Quine, le résultat du consensus appartient déjà à F. Donc, t implique un terme de F. C.Q.F.D.

Une conséquence de ce que nous venons de démontrer est que la formule F contient tous les impliquants premiers de F. De plus, elle ne peut contenir aucun autre impliquant de F puisqu'aucune simplification ne peut s'y appliquer.

À titre d'exemple, calculons la liste des impliquants premiers de la formule $F = \overline{a}\,\overline{c} + a\overline{b} + \overline{a}b + ac + bc$, en utilisant cette méthode. Il n'y a pas de simplification possible, puisque aucun terme n'en implique un autre. On recherche donc les consensus applicables. Le consensus des termes $\overline{a}\,\overline{c}$ et $a\overline{b}$ produit le nouveau terme $\overline{b}\overline{c}$. La liste courante devient $\overline{a}\,\overline{c} + \overline{b}\overline{c} + a\overline{b} + \overline{a}b + ac + bc$.

Aucune simplification n'est possible pour cette formule et les consensus possibles ne donnent pas de nouveaux termes. La liste des impliquants premiers est donc $\overline{a}\,\overline{c} + \overline{b}\overline{c} + a\overline{b} + \overline{a}b + ac + bc$.

Cette seconde méthode de Quine est élégante. Néanmoins, l'examen de la formule, pour détecter les termes qui en impliquent d'autres, est fastidieuse et c'est encore plus le cas pour la détection des consensus qui ont déjà été effectués (ou non). Nous verrons à la section 1.3.5 une autre méthode qui applique les mêmes principes mais est plus systématique.

1.3.3 Méthode de McCluskey

La méthode de McCluskey [McC56] est une systématisation de la première méthode de Quine. McCluskey utilise notamment deux représentations des termes qui sont pratiques pour expliquer le calcul des impliquants, mais ne sont pas indispensables à la compréhension de la méthode. Nous présentons tout d'abord la méthode de "base" en utilisant nos notations. Nous discutons ensuite certaines optimisations de cette méthode, en faisant référence à ces représentations là où elles sont utiles.

Méthode de base

Nous rappelons que la méthode de Quine part d'une formule canonique qui est une liste de termes de m lettres, où m est le nombre de lettres de la formule. A partir de cette liste, on calcule d'abord les impliquants de taille $m-1$; puis ceux de taille $m-2$, à partir des impliquants de

taille $m-1$. On itère le processus jusqu'à ce qu'il n'y ait plus de termes fusionnables. On maintient aussi une liste de termes contenant les impliquants premiers déjà identifiés. On marque à chaque étape les termes qui participent à une fusion, puis on ajoute les termes non marqués à la liste des impliquants premiers.

L'amélioration de McCluskey se situe dans la façon dont on détermine, à chaque étape, les nouveaux impliquants à partir des anciens. Chez Quine, il n'y a pas de précisions apportées à la façon de mettre en relation les termes fusionnables. Donc les "anciens" impliquants sont comparés chacun à chacun. McCluskey propose de les classer en fonction de leur nombre de littéraux positifs (et donc aussi négatifs).

À l'étape $m-l+1$ (calculer les termes de taille $l-1$ à partir des termes de taille l), on divise l'ensemble des termes de taille l, en $l+1$ listes $li(0),\ldots,li(l)$ comportant respectivement $0,\ldots,l$ littéraux positifs et on compare les termes des listes $li(x)$ et $li(x+1)$ pour déterminer les listes $li'(x)$ contenant les impliquants de taille $l-1$ contenant x littéraux positifs.

Prenons par exemple la formule canonique $F = \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + \overline{a}b\overline{c}d + ab\overline{c}d + \overline{a}\overline{b}cd + a\overline{b}cd + \overline{a}bcd + abcd$. La liste initiale des impliquants est formée des termes de F.

À l'étape 1, les 5 listes d'impliquants sont :

0 `littéraux positifs` : $\overline{a}\overline{b}\overline{c}\overline{d}$
1 `littéral positif` : $\overline{a}\overline{b}c\overline{d}$
2 `littéraux positifs` : $\overline{a}b\overline{c}d + \overline{a}\overline{b}cd$
3 `littéraux positifs` : $ab\overline{c}d + a\overline{b}cd + \overline{a}bcd$
4 `littéraux positifs` : $abcd$

On effectue les fusions présentées à la table 1.4.

0/1	$\overline{a}\overline{b}c\overline{d}$
$\overline{a}\overline{b}\overline{c}\overline{d}$	$\overline{a}\overline{b}\,\overline{d}$

(a) $li'(0)$

1/2	$\overline{a}b\overline{c}d$	$\overline{a}\overline{b}cd$
$\overline{a}\overline{b}c\overline{d}$	/	$\overline{a}\overline{b}c$

(b) $li'(1)$

2/3	$ab\overline{c}d$	$a\overline{b}cd$	$\overline{a}bcd$
$\overline{a}b\overline{c}d$	$b\overline{c}d$	/	$\overline{a}bd$
$\overline{a}\overline{b}cd$	/	$\overline{b}cd$	$\overline{a}cd$

(c) $li'(2)$

3/4	$abcd$
$ab\overline{c}d$	abd
$a\overline{b}cd$	acd
$\overline{a}bcd$	bcd

(d) $li'(3)$

Table 1.4 – Calcul des listes impliquants de taille 3 de G

Tous les termes étant fusionnables à un autre terme, aucun terme n'est

placé dans la liste des impliquants premiers. On passe à l'étape suivante.
À l'étape 2, les 4 listes d'impliquants sont :

$$\begin{array}{lll} 0 & \texttt{littéraux positifs} & : \quad \overline{a}\overline{b}\,\overline{d} \\ 1 & \texttt{littéral positif} & : \quad \overline{a}\overline{b}c \\ 2 & \texttt{littéraux positifs} & : \quad \overline{a}bd + b\overline{c}d + \overline{a}cd + \overline{b}cd \\ 3 & \texttt{littéraux positifs} & : \quad abd + acd + bcd \end{array}$$

On obtient les fusions présentées à la table 1.5.

0/1	$\overline{a}\overline{b}c$
$\overline{a}\overline{b}\,\overline{d}$	/

(a) $li'(0)$

1/2	$\overline{a}bd$	$b\overline{c}d$	$\overline{a}cd$	$\overline{b}cd$
$\overline{a}\overline{b}c$	/	/	/	/

(b) $li'(1)$

2/3	abd	acd	bcd
$\overline{a}bd$	bd	/	/
$b\overline{c}d$	/	/	bd
$\overline{a}cd$	/	cd	/
$\overline{b}cd$	/	/	cd

(c) $li'(2)$

Table 1.5 – Calcul des listes impliquants de taille 2 de G

Les termes non fusionnables, c'est-à-dire : non marqués à cette étape, sont $\overline{a}\overline{b}\,\overline{d}$ et $\overline{a}\overline{b}c$. Ils sont placés dans la liste des impliquants premiers. On passe à l'étape suivante.
À l'étape 3, les 3 listes d'impliquants sont :

$$\begin{array}{lll} 0 & \texttt{littéraux positifs} & : \quad / \\ 1 & \texttt{littéral positif} & : \quad / \\ 2 & \texttt{littéraux positifs} & : \quad bd + cd \end{array}$$

Il n'y a pas de fusions possibles, puisque les listes $li(0)$ et $li(1)$ sont vides. Les termes bd et cd sont les impliquants premiers trouvés à cette étape. La liste des impliquants premiers est donc $\overline{a}\overline{b}\,\overline{d} + \overline{a}\overline{b}c + bd + cd$.

Amélioration 1

On peut systématiser davantage. McCluskey propose une méthode pour ne calculer qu'une seule fois chaque impliquant de taille $l-1$ à partir des impliquants de taille l. C'est là qu'intervient la représentation des termes par des entiers.

Chaque terme complet a une représentation binaire obtenue en remplaçant chaque littéral négatif par un 0 et chaque littéral positif par un 1. Cette représentation définit un nombre entier non négatif qui identifie le terme complet. Les termes complets $\overline{a}b\overline{c}d$, $ab\overline{c}d$, $\overline{a}bcd$ et $abcd$ sont respectivement identifiés par les entiers 10 (1010 en binaire), 11 (1011 en

binaire), 14 (1110 en binaire) et 15 (1111 en binaire). Un terme incomplet peut alors être identifié par la liste triée des entiers qui identifient les termes complets impliquant le terme incomplet considéré. Ainsi, le terme bd dont la suite de termes complets est $\overline{a}b\overline{c}d + ab\overline{c}d + \overline{a}bcd + abcd$, sera identifié par la liste d'entiers $\langle 10, 11, 14, 15\rangle$.

McCluskey propose de calculer chaque impliquant de taille $l-1$ en mettant en correspondance les deux impliquants de taille l dont la concaténation des listes triées d'entiers donne exactement la liste triée de l'impliquant de taille $l-1$.

Si on range les termes dans l'ordre lexicographique de leur liste d'entiers, on peut limiter le nombre de couples de termes (t_1, t_2) à comparer : ayant calculé la liste de t_1, on sait que le premier entier de la liste de t_2, doit être strictement supérieur au dernier entier de la liste de t_1. Donc, on peut commencer la recherche dans la liste contenant t_2, à partir d'un terme associé à une telle liste, en utilisant une recherche dichotomique, par exemple.

Nous appliquons cette méthode aussi à la formule canonique F, choisie comme exemple, un peu plus haut. Les représentations entières de ses termes complets $\overline{a}\overline{b}\overline{c}\overline{d}$, $\overline{a}\overline{b}c\overline{d}$, $\overline{a}b\overline{c}d$, $ab\overline{c}d$, $\overline{a}\overline{b}cd$, $a\overline{b}cd$, $\overline{a}bcd$ et $abcd$ sont respectivement $0, 4, 10, 11, 12, 13, 14$ et 15.

Les fusions de l'étape 1 sont alors :

$li(0)$ `contre` $li(1)$

$0 : \overline{a}\overline{b}\overline{c}\overline{d}$ `contre` $4 : \overline{a}\overline{b}c\overline{d} \Longrightarrow \langle 0, 4\rangle : \overline{a}\overline{b}\,\overline{d}$

$li(1)$ `contre` $li(2)$

$4 : \overline{a}\overline{b}c\overline{d}$ `contre` $10 : \overline{a}b\overline{c}d \Longrightarrow /$

$4 : \overline{a}\overline{b}c\overline{d}$ `contre` $12 : \overline{a}\overline{b}cd \Longrightarrow \langle 4, 12\rangle : \overline{a}\overline{b}c$

$li(2)$ `contre` $li(3)$

$10 : \overline{a}b\overline{c}d$ `contre` $11 : ab\overline{c}d \Longrightarrow \langle 10, 11\rangle : b\overline{c}d$

$10 : \overline{a}b\overline{c}d$ `contre` $13 : a\overline{b}cd \Longrightarrow /$

$10 : \overline{a}b\overline{c}d$ `contre` $14 : \overline{a}bcd \Longrightarrow \langle 10, 14\rangle : \overline{a}bd$

$12 : \overline{a}\overline{b}cd$ `contre` $13 : a\overline{b}cd \Longrightarrow \langle 12, 13\rangle : \overline{b}cd$

$12 : \overline{a}\overline{b}cd$ `contre` $14 : \overline{a}bcd \Longrightarrow \langle 12, 14\rangle : \overline{a}cd$

$li(3)$ `contre` $li(4)$

$11 : ab\overline{c}d$ `contre` $15 : abcd \Longrightarrow \langle 11, 15\rangle : abd$

$13 : a\overline{b}cd$ `contre` $15 : abcd \Longrightarrow \langle 13, 15\rangle : acd$

$14 : \overline{a}bcd$ `contre` $15 : abcd \Longrightarrow \langle 14, 15\rangle : bcd$

On passe à l'étape 2. Les fusions de cette étape sont :

$li(0)$ `contre` $li(1)$

$\langle 0, 4\rangle : \overline{a}\overline{b}\,\overline{d}$ `contre` $/ \Longrightarrow /$

$li(1)$ `contre` $li(2)$

$\langle 4, 12\rangle : \overline{a}\overline{b}c$ `contre` $/ \Longrightarrow /$

$li(2)$ `contre` $li(3)$

$\langle 10,11\rangle : b\overline{c}d$ `contre` $\langle 13,15\rangle : acd \implies /$

$\langle 10,11\rangle : b\overline{c}d$ `contre` $\langle 14,15\rangle : bcd \implies \langle 10,11,14,15\rangle : bd$

$\langle 12,13\rangle : \overline{b}cd$ `contre` $\langle 14,15\rangle : bcd \implies \langle 12,13,14,15\rangle : cd$

L'implémentation systématique de cette méthode pose quelques difficultés (car on ne veut pas représenter explicitement les listes d'entiers associées aux termes). Nous en parlons plus loin dans la section 1.4 sur l'implémentation.

Il y a encore un autre problème avec cette méthode : tous les termes fusionnables à un autre terme ne sont pas nécessairement marqués. Ainsi, dans l'exemple précédent, seuls les termes $b\overline{c}d$, $\overline{b}cd$, et bcd sont marqués, à l'étape 2, car ils participent effectivement à une fusion. Les termes $\overline{a}bd$, $\overline{a}cd$, abd et acd ne sont pas marqués, pourtant ils sont fusionnables, chacun avec un autre, et ne sont donc pas des impliquants premiers.

McCluskey ne propose pas de méthode précise pour éviter ce problème. Cependant, on peut déterminer tous les termes fusionnables donnant (potentiellement) lieu à un impliquant à partir de cet impliquant lui-même, en lui ajoutant deux littéraux opposés portant sur la même lettre et ce, pour toutes les lettres qui n'y figurent pas. Une solution est donc de calculer ces termes fusionnables et de les marquer "*a posteriori*". Ainsi, les termes $\overline{a}bd$, $\overline{a}cd$, abd et acd peuvent être retrouvés en ajoutant les littéraux $\overline{a}$ et a, aux impliquants premiers bd et cd. Ce marquage *a posteriori* contrebalance défavorablement l'avantage obtenu en ne calculant qu'une fois chaque impliquant.

Amélioration 2

Vu le problème ci-dessus, on peut procéder différemment, en calculant exactement une fois toutes les fusions possibles : on fait des sous-listes des listes $li(x)$ et $li(x+1)$ correspondant aux termes qui contiennent la même lettre l, mais sous forme négative ($\overline{l}$) dans $li(x)$ et positive dans $li(x+1)$. Ces listes étant triées, on peut les parcourir en parallèle (efficacement). Il faut ensuite fusionner les listes résultantes pour que chaque impliquant ne soit représenté qu'une fois.

Nous appliquons également cette méthode à la formule F. Les fusions de l'étape 1 sont alors :

$li(0)$ `contre` $li(1)$

$\overline{a} : \overline{a}\overline{b}\overline{c}\overline{d}$ `contre` $a : / \implies /$

$\overline{b} : \overline{a}\overline{b}\overline{c}\overline{d}$ `contre` $b : / \implies /$

$\overline{c} : \overline{a}\overline{b}\overline{c}\overline{d}$ `contre` $c : \overline{a}\overline{b}c\overline{d} \implies \overline{a}\overline{b}\,\overline{d}$

$\overline{d} : \overline{a}\overline{b}\overline{c}\overline{d}$ `contre` $d : / \implies /$

$li(1)$ `contre` $li(2)$

$\overline{a} : \overline{a}\overline{b}c\overline{d}$ `contre` $a : /$ $\Longrightarrow$ $/$
$\overline{b} : \overline{a}\overline{b}c\overline{d}$ `contre` $b : \overline{a}b\overline{c}d$ $\Longrightarrow$ $/$
$\overline{c} : /$ `contre` $c : \overline{a}\overline{b}cd$ $\Longrightarrow$ $/$
$\overline{d} : \overline{a}\overline{b}c\overline{d}$ `contre` $d : \overline{a}b\overline{c}d + \overline{a}\overline{b}cd$ $\Longrightarrow$ $\overline{a}\overline{b}c$

$li(2)$ `contre` $li(3)$

$\overline{a} : \overline{a}b\overline{c}d + \overline{a}\overline{b}cd$ `contre` $a : ab\overline{c}d + a\overline{b}cd$ $\Longrightarrow$ $b\overline{c}d + \overline{b}cd$
$\overline{b} : \overline{a}\overline{b}cd$ `contre` $b : ab\overline{c}d + \overline{a}bcd$ $\Longrightarrow$ $\overline{a}cd$
$\overline{c} : \overline{a}b\overline{c}d$ `contre` $c : a\overline{b}cd + \overline{a}bcd$ $\Longrightarrow$ $\overline{a}bd$
$\overline{d} : /$ `contre` $d : ab\overline{c}d + a\overline{b}cd + \overline{a}bcd$ $\Longrightarrow$ $/$

$li(3)$ `contre` $li(4)$

$\overline{a} : \overline{a}bcd$ `contre` $a : abcd$ $\Longrightarrow$ bcd
$\overline{b} : a\overline{b}cd$ `contre` $b : abcd$ $\Longrightarrow$ acd
$\overline{c} : ab\overline{c}d$ `contre` $c : abcd$ $\Longrightarrow$ abd
$\overline{d} : /$ `contre` $d : abcd$ $\Longrightarrow$ $/$

On passe à l'étage 2. Les fusions de cette étape sont :

$li(0)$ `contre` $li(1)$

$\overline{a} : \overline{a}\overline{b}\,\overline{d}$ `contre` $a : /$ $\Longrightarrow$ $/$
$\overline{b} : \overline{a}\overline{b}\,\overline{d}$ `contre` $b : /$ $\Longrightarrow$ $/$
$\overline{c} : /$ `contre` $c : \overline{a}\overline{b}c$ $\Longrightarrow$ $/$
$\overline{d} : \overline{a}\overline{b}\,\overline{d}$ `contre` $d : /$ $\Longrightarrow$ $/$

$li(1)$ `contre` $li(2)$

$\overline{a} : \overline{a}\overline{b}c$ `contre` $a : /$ $\Longrightarrow$ $/$
$\overline{b} : \overline{a}\overline{b}c$ `contre` $b : \overline{a}bd + b\overline{c}d$ $\Longrightarrow$ $/$
$\overline{c} : /$ `contre` $c : \overline{a}cd + \overline{b}cd$ $\Longrightarrow$ $/$
$\overline{d} : /$ `contre` $d : \overline{a}bd + b\overline{c}d + \overline{a}cd + \overline{b}cd$ $\Longrightarrow$ $/$

$li(2)$ `contre` $li(3)$

$\overline{a} : \overline{a}bd + \overline{a}cd$ `contre` $a : abd + acd$ $\Longrightarrow$ $bd + cd$
$\overline{b} : \overline{b}cd$ `contre` $b : abd + bcd$ $\Longrightarrow$ cd
$\overline{c} : b\overline{c}d$ `contre` $c : acd + bcd$ $\Longrightarrow$ bd
$\overline{d} : /$ `contre` $d : abd + acd + bcd$ $\Longrightarrow$ $/$

1.3.4 Méthode de Coudert

Dans l'article [Cou94], Olivier Coudert propose une méthode pour calculer l'ensemble des impliquants premiers d'une formule canonique, utilisant une stratégie du type "diviser pour régner" et propice à l'utilisation de structures de données compactes dérivées des BDDs [Bry86, Min93b]. L'utilisation de ces structures est fondamentale pour améliorer l'efficacité pratique de l'algorithme proposé mais n'est pas idéale pour faciliter sa compréhension. Coudert lui-même présente sa méthode en traitant les formules comme des ensembles de termes, ce qui permet de la

comprendre plus aisément. Comme nos conventions terminologiques reviennent aussi à considérer les formules comme des ensembles (en fait, des suites strictement triées) de termes, nous présentons sa méthode en la réexprimant dans nos notations, les mêmes que celles des sections précédentes. Nous expliquons et justifions sa méthode avec plus de détails que dans son article de référence.

Notations supplémentaires

Soient F et G, deux formules. Nous définissons $F \cup G$, $F \cap G$, $F \setminus G$ comme étant égales aux formules dont l'ensemble des termes est, selon le cas, l'*union*, l'*intersection*, et la *différence* des ensembles de termes de F et G. Avec nos représentations des formules, ces formules peuvent être calculées en parcourant les formules F et G, une seule fois, en parallèle.

La méthode de calcul des impliquants premiers de Coudert est essentiellement basée sur, et justifiée par, le théorème ci-dessous.

Théorème 1. *Soit F, une formule canonique. Soient L, l'ensemble des lettres utilisées par F et $l \in L$.*

Il existe donc des formules canoniques F_0 et F_1, utilisant les lettres de $L \setminus \{l\}$, telles que $F = \overline{l}.F_0 \cup l.F_1$. Soient IP, IP_0, IP_1, IP_*, les ensembles d'impliquants premiers de F, F_0, F_1, et $F_0 \cap F_1$, respectivement.

On a, alors :

$$IP = IP_* \cup l.(IP_1 \setminus IP_*) \cup \overline{l}.(IP_0 \setminus IP_*).$$

Pour simplifier la preuve de ce théorème, nous allons démontrer quatre lemmes. Pour prouver ceux-ci nous "voyons" les formules comme des ensembles de termes et nous associons à chaque terme t, n'utilisant que des lettres figurant dans l'ensemble de lettres L, la formule canonique $\mathtt{FC}(t, L)$ logiquement équivalente à t composée uniquement de termes utilisant toutes les lettres de L. Par exemple, si $t = \overline{a}cd$ et $L = \{a, b, c, d, e\}$, on a $\mathtt{FC}(t, L) = \overline{a}\overline{b}cd\overline{e} + \overline{a}bcd\overline{e} + \overline{a}\overline{b}cde + \overline{a}bcde$. Ceci nous permet de raisonner en remplaçant la notion d'implication logique entre termes et formules, par la notion d'inclusion entre formules (canoniques et utilisant les mêmes lettres). On se base en particulier sur le fait que t est un impliquant premier de la formule canonique F, utilisant l'ensemble de lettres L, si et seulement si les deux conditions ci-dessous sont vérifiées :

$$\begin{aligned} &\mathtt{FC}(t, L) \subseteq F \\ &\mathtt{FC}(t', L) \not\subseteq F, \qquad \text{si } t' \text{ est un sous-terme strict de } t. \end{aligned}$$

Les énoncés des lemmes suivants utilisent les notations du théorème 1.

Lemme 1. *$l.t$ est un impliquant premier de F si et seulement si t est un impliquant premier de F_1 et n'est pas un impliquant de $F_0 \cap F_1$.*

Lemme 2. *$\overline{l}.t$ est un impliquant premier de F si et seulement si t est un impliquant premier de F_0 et n'est pas un impliquant de $F_0 \cap F_1$.*

Lemme 3. *Si t ne contient pas l, c'est un impliquant premier de F, si et seulement si c'est un impliquant premier de $F_0 \cap F_1$.*

Preuve du lemme 1. Nous réécrivons l'énoncé en terme de relations d'inclusions entre formules canoniques.
D'abord, l'énoncé "$l.t$ est un impliquant premier de F" équivaut aux trois conditions :

$$\texttt{FC}(l.t, L) \subseteq F \tag{1.1}$$
$$\texttt{FC}(l.t', L) \not\subseteq F, \qquad \text{pour tout } t', \text{ sous-terme strict de } t \tag{1.2}$$
$$\texttt{FC}(t, L) \not\subseteq F \tag{1.3}$$

(En effet, il n'est pas nécessaire d'ajouter que $\texttt{FC}(t', L) \not\subseteq F$ (1.3') pour les sous-termes strict de t, puisqu'on a, dans ce cas, $\texttt{FC}(t, L) \subseteq \texttt{FC}(t', L)$ (1.3") et que (1.3) et (1.3") impliquent (1.3').)
Ensuite, l'énoncé "t est un impliquant premier de F_1 et n'est pas un impliquant de $F_0 \cap F_1$" équivaut aux trois autres conditions :

$$\texttt{FC}(t, L \setminus \{l\}) \subseteq F_1 \tag{1.4}$$
$$\texttt{FC}(t', L \setminus \{l\}) \not\subseteq F_1, \qquad \text{pour tout } t', \text{ sous-terme strict de } t \tag{1.5}$$
$$\texttt{FC}(t, L \setminus \{l\}) \not\subseteq F_0 \cap F_1 \tag{1.6}$$

Nous montrons maintenant que les trois premières conditions sont équivalentes aux trois dernières, chacune à chacune.

- Les conditions (1.1) et (1.4) sont équivalentes puisque $\texttt{FC}(l.t, L) = l.\texttt{FC}(t, L \setminus \{l\})$ et $F = \overline{l}.F_0 \cup l.F_1$.
- La condition $\texttt{FC}(l.t', L) \not\subseteq F$, équivaut à $l.\texttt{FC}(t', L \setminus \{l\}) \not\subseteq \overline{l}.F_0 \cup l.F_1$, qui équivaut à $\texttt{FC}(t', L \setminus \{l\}) \not\subseteq F_1$, pour tout terme t', ne contenant que des lettres de $L \setminus \{l\}$. Par conséquent (1.2) et (1.5) sont équivalentes.
- La condition $\texttt{FC}(t, L) \not\subseteq F$ équivaut à $\overline{l}.\texttt{FC}(t, L \setminus \{l\}) \cup l.\texttt{FC}(t, L \setminus \{l\}) \not\subseteq \overline{l}.F_0 \cup l.F_1$, qui équivaut à $(\overline{l}.\texttt{FC}(t, L \setminus \{l\}) \not\subseteq \overline{l}.F_0)$ ou $(l.\texttt{FC}(t, L \setminus \{l\}) \not\subseteq l.F_1)$, qui équivaut à $(\texttt{FC}(t, L \setminus \{l\}) \not\subseteq F_0)$ ou $(\texttt{FC}(t, L \setminus \{l\}) \not\subseteq F_1)$, qui équivaut enfin à $\texttt{FC}(t, L \setminus \{l\}) \not\subseteq F_0 \cap F_1$. □

La preuve du lemme 2 est comme celle du lemme 1, en échangeant les rôles des littéraux l et $\overline{l}$.

Preuve du lemme 3. Soit t qui ne contient pas la lettre l. L'énoncé "t est un impliquant premier de F", équivaut aux deux conditions :

$$\texttt{FC}(t, L) \subseteq F \tag{1.7}$$
$$\texttt{FC}(t', L) \not\subseteq F, \qquad \text{pour tout } t', \text{ sous-terme strict de } t \tag{1.8}$$

Ensuite, l'énoncé "t est un impliquant premier de $F_0 \cap F_1$" équivaut aux deux conditions :

$$\texttt{FC}(t, L \setminus \{l\}) \subseteq F_0 \cap F_1 \tag{1.9}$$
$$\texttt{FC}(t', L \setminus \{l\}) \not\subseteq F_0 \cap F_1, \quad \text{pour tout } t', \text{ sous-terme strict de } t \tag{1.10}$$

Montrons que les conditions (1.7) et (1.8) sont équivalentes, respectivement, à (1.9) et (1.10).

- (1.7) équivaut à $\overline{l}.\texttt{FC}(t, L \setminus \{l\}) \cup l.\texttt{FC}(t, L \setminus \{l\}) \subseteq \overline{l}.F_0 \cup l.F_1$, qui équivaut à $\overline{l}.\texttt{FC}(t, L \setminus \{l\}) \subseteq \overline{l}.F_0$ et $l.\texttt{FC}(t, L \setminus \{l\}) \subseteq l.F_1$, qui équivaut à $\texttt{FC}(t, L \setminus \{l\}) \subseteq F_0$ et $\texttt{FC}(t, L \setminus \{l\}) \subseteq F_1$, qui équivaut à (1.9).
- Soit t' un sous-terme strict de t. La condition $\texttt{FC}(t', L) \not\subseteq F$ équivaut à $\overline{l}.\texttt{FC}(t', L \setminus \{l\}) \cup l.\texttt{FC}(t', L \setminus \{l\}) \not\subseteq \overline{l}.F_0 \cup l.F_1$, qui équivaut à $\overline{l}.\texttt{FC}(t', L \setminus \{l\}) \not\subseteq \overline{l}.F_0$ ou $l.\texttt{FC}(t', L \setminus \{l\}) \not\subseteq l.F_1$, qui équivaut à $(\texttt{FC}(t', L \setminus \{l\}) \not\subseteq F_0)$ ou $(\texttt{FC}(t', L \setminus \{l\}) \not\subseteq F_1)$, qui équivaut à $\texttt{FC}(t', L \setminus \{l\}) \not\subseteq F_0 \cap F_1$. Par conséquent les conditions (1.8) et (1.10) sont équivalentes. □

Lemme 4. *Si t est un impliquant premier de F et si $G \subseteq F$, t n'est pas un impliquant non premier de G.*

Preuve du lemme 4. Supposons que t soit un impliquant non premier de G. Alors, il existe t', sous-terme strict de t tel que $\texttt{FC}(t', L) \subseteq G$. Comme $G \subseteq F$, on a aussi $\texttt{FC}(t', L) \subseteq F$, ce qui est impossible puisque t est un impliquant premier de F. □

Preuve du théorème principal.

a) IP_* est égal à l'ensemble des impliquants premiers de F qui ne contiennent pas la lettre l. C'est une conséquence immédiate du lemme 3.

b) $l.(IP_1 \setminus IP_*)$ est égal à l'ensemble des impliquants premiers de F qui contiennent le littéral l. En effet, le lemme 1 nous assure que l'ensemble des impliquants premiers de F qui contiennent le littéral l est égal à $l.(IP_1 \setminus I_*)$, où I_* est l'ensemble de tous les impliquants de $F_0 \cap F_1$ (non nécessairement premiers). Heureusement, le lemme 4, nous assure qu'aucun impliquant non premier

de $F_0 \cap F_1$ n'est un impliquant premier de F_1. Il s'en suit que $IP_1 \setminus I_* = IP_1 \setminus IP_*$. D'où le résultat demandé.

c) Par un raisonnement semblable ou démontre que $\overline{l}.(IP_0 \setminus IP_*)$ est égal à l'ensemble des impliquants premiers de F qui contiennent le littéral $\overline{l}$.

d) La preuve du théorème découle alors immédiatement des points a), b) et c). □

Nous allons maintenant énoncer l'algorithme de calcul des impliquants premiers d'une formule canonique F, proposé par Coudert dans [Cou94]. L'algorithme est récursif. Pour comprendre correctement ses deux cas de base, il faut se rappeler que 0 représente une suite vide de termes. Par contre, la formule 1, correspond à une suite composée d'un seul terme utilisant un ensemble vide de lettres.

Soit F, une formule canonique. On calcule ses impliquants premiers comme suit :

1. Si $F = 0$, on renvoie 0.
2. Si $F = 1$, on renvoie 1.
3. Sinon, on choisit une lettre l utilisée par F et on détermine les formules canoniques F_0 et F_1, n'utilisant pas l, telles que $F = \overline{l}.F_0 \cup l.F_1$.
   On calcule alors récursivement IP_0, IP_1 et IP_*, en appliquant l'algorithme aux formules F_0, F_1 et $F_0 \cap F_1$, respectivement.
   Finalement, on renvoie la formule
   $$IP_* \cup \overline{l}.(IP_0 \setminus IP_*) \cup l.(IP_1 \setminus IP_*).$$

Preuve de correction de l'algorithme.

La correction de l'algorithme se prouve par induction sur le nombre de lettres utilisées par la formule F.

Si $F = 0$, le résultat est correct puisque 0 ne possède aucun impliquant. Donc, 0 est correct puisque c'est un ensemble vide de termes. Si $F = 1$, le résultat est également correct puisque 1 est le seul impliquant de 1.

Sinon, on peut supposer, par hypothèse d'induction, que les formules IP_0, IP_1 et IP_* renvoyées par l'algorithme sont les listes d'impliquants premiers de F_0, F_1 et $F_0 \cap F_1$, respectivement. Le résultat renvoyé par l'algorithme pour F est donc correct, en vertu du théorème démontré plus haut. □

À titre d'exemple, nous appliquons l'algorithme ci-dessus au calcul des impliquants premiers de la formule $F = \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + \overline{a}b\overline{c}d + ab\overline{c}d + \overline{a}\overline{b}cd + a\overline{b}cd + \overline{a}bcd + abcd$.

Pour simplifier la présentation, nous introduisons trois “optimisations” :

1. Si F contient au plus 1 terme, on renvoie directement F.
2. Si F contient 2^m termes, où m est le nombre de lettres de ses termes, on renvoie directement 1.
3. Si l'algorithme est exécuté plusieurs fois avec la même (sous-)formule, on ne développe que la première exécution mais on écrit une égalité de la forme $F_i = F_j$ pour exprimer que le “nouvel appel” de l'algorithme, avec la formule F_i se déroule comme, et produit le même résultat que, l'appel avec la formule F_j, développé précédemment.

Voici la trace de l'exécution de l'algorithme, avec ces conventions :

$$
\begin{aligned}
F_0 &= \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + \overline{a}b\overline{c}\overline{d} + ab\overline{c}\overline{d} + \overline{a}\overline{b}cd + a\overline{b}cd + \overline{a}bcd + abcd \\
&= \overline{d}.F_1 + d.F_2 \\
&\quad F_1 \cap F_2 = F_3 \\
&\quad F_1 = \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c \\
&\qquad = \overline{c}.F_4 + c.F_5 \\
&\qquad\quad F_4 \cap F_5 = F_6 \\
&\qquad\quad F_4 = \overline{a}\overline{b} \\
&\qquad\qquad = \mathit{IP}_4 \\
&\qquad\quad F_5 = \overline{a}\overline{b} \\
&\qquad\qquad = \mathit{IP}_5 \\
&\qquad\quad F_6 = \overline{a}\overline{b} \\
&\qquad\qquad = \mathit{IP}_6 \\
&\quad \mathit{IP}_1 = \mathit{IP}_6 + \overline{c}.(\mathit{IP}_4 \setminus \mathit{IP}_6) + c.(\mathit{IP}_5 \setminus \mathit{IP}_6) \\
&\qquad = \overline{a}\overline{b} + \overline{c}.(\overline{a}\overline{b} \setminus \overline{a}\overline{b}) + c.(\overline{a}\overline{b} \setminus \overline{a}\overline{b}) \\
&\qquad = \overline{a}\overline{b} \\
&\quad F_2 = \overline{a}b\overline{c} + ab\overline{c} + \overline{a}\overline{b}c + a\overline{b}c + \overline{a}bc + abc \\
&\qquad = \overline{c}.F_7 + c.F_8 \\
&\qquad\quad F_7 \cap F_8 = F_9 \\
&\qquad\quad F_7 = \overline{a}b + ab \\
&\qquad\qquad = \overline{b}.F_{10} + b.F_{11} \\
&\qquad\qquad\quad F_{10} \cap F_{11} = F_{12} \\
&\qquad\qquad\quad F_{10} = 0 \\
&\qquad\qquad\qquad = \mathit{IP}_{10} \\
&\qquad\qquad\quad F_{11} = \overline{a} + a \\
&\qquad\qquad\quad \mathit{IP}_{11} = 1 \ (\texttt{car } F_{11} \texttt{ contient } 2^1 \texttt{ termes}) \\
&\qquad\qquad\quad F_{12} = 0 \\
&\qquad\qquad\qquad = \mathit{IP}_{12} \\
&\qquad\quad \mathit{IP}_7 = \mathit{IP}_{12} + \overline{b}.(\mathit{IP}_{10} \setminus \mathit{IP}_{12}) + b.(\mathit{IP}_{11} \setminus \mathit{IP}_{12}) \\
&\qquad\qquad = 0 + \overline{b}.(0 \setminus 0) + b.(1 \setminus 0) \\
&\qquad\qquad = b \\
&\qquad\quad F_8 = \overline{a}\overline{b} + a\overline{b} + \overline{a}b + ab \\
&\qquad\quad \mathit{IP}_8 = 1 \ (\texttt{car } F_8 \texttt{ contient } 2^2 \texttt{ termes})
\end{aligned}
$$

$$
\begin{aligned}
F_9 &= F_7 \\
IP_2 &= IP_9 + \overline{c}.(IP_7 \setminus IP_9) + c.(IP_8 \setminus IP_9) \\
&= b + \overline{c}.(b \setminus b) + c.(1 \setminus b) \\
&= b + c \\
F_3 &= a\overline{b}c \\
&= IP_3 \\
IP_0 &= IP_3 + \overline{d}.(IP_1 \setminus IP_3) + d.(IP_2 \setminus IP_3) \\
&= a\overline{b}c + \overline{d}.(a\overline{b} \setminus a\overline{b}c) + d.((b+c) \setminus a\overline{b}c) \\
&= a\overline{b}\,\overline{d} + a\overline{b}c + bd + cd
\end{aligned}
$$

1.3.5 Méthode de Thayse

Thayse [Tha04] propose un formalisme général pour résoudre les problèmes modélisables grâce au calcul booléen. Parmi d'autres applications de sa théorie, il décrit une méthode pour calculer la liste des impliquants premiers d'une formule booléenne, basée sur la notion de dérivée d'une formule par rapport à un ensemble de lettres. Nous donnons une présentation simplifiée de cette notion, suffisante pour comprendre et justifier la correction de sa méthode.

Deux nouvelles opérations sur les formules

Nous introduisons d'abord deux nouvelles opérations sur les formules, écrites selon nos notations, qui sont les primitives de base de son algorithme.

Soit F, une formule. Nous notons $\texttt{absorber}(F)$ la formule obtenue en retirant de F tous les termes qui en impliquent un autre. Par exemple, si $F = a + a\overline{b} + bc$, $\texttt{absorber}(F) = a + bc$.

La seconde opération, appelée *conjonction formelle* et notée " $\hat{.}$ ", est définie comme suit. Soient t_1 et t_2, deux termes. Nous disons qu'ils sont joignables s'ils ne contiennent pas de littéraux contradictoires (portant donc sur la même lettre). Par exemple, les termes $a\overline{c}d$ et bd sont joignables, les termes $a\overline{c}$ et bcd ne le sont pas. Si t_1 et t_2 sont joignables, nous appelons jointure de t_1 et t_2, le terme ayant pour littéraux l'ensemble des littéraux de t_1 et t_2. Soient alors F et G, deux formules. On appelle conjonction formelle de F et G, la formule, notée $F \hat{.} G$, formée de tous les termes obtenus en choisissant deux termes joignables t_1 et t_2, respectivement dans F et G, et en effectuant leur jointure. Par exemple, si $F = a\overline{c} + bd$ et $G = c + ad$, on a : $F \hat{.} G = a\overline{c}d + abd + bcd$. On voit également facilement que la formule $F \hat{.} G$ est logiquement équivalente à la conjonction des formules F et G, au sens usuel du terme.

Dérivées d'une formule booléenne

Nous introduisons maintenant deux opérations de dérivation sur les formules.

Soit F une formule se décomposant sous la forme habituelle : $\overline{l}.F_0 \cup l.F_1 \cup F_*$. On appelle *dérivée de F par rapport à l*, et on note $\mathtt{D}_l(F)$, la formule égale à

$$(F_0 \cup F_*) \widehat{.} (F_1 \cup F_*).$$

Par exemple, $\mathtt{D}_a(\overline{a}\overline{b}\overline{c} + a + b\overline{c}) = (\overline{b}\overline{c} + b\overline{c}) \widehat{.} (1 + b\overline{c}) = \overline{b}\overline{c} + b\overline{c}$.

Nous définissons ensuite la notion de *dérivée de F par rapport à un ensemble de lettres L*, notée $\mathtt{D}_L(F)$.

1. Si $L = \{\}$, par définition, $\mathtt{D}_L(F) = F$.
2. Si L n'est pas vide, soit $l \in L$. Par définition,

$$\mathtt{D}_L(F) = \mathtt{absorber}(G \cup \mathtt{D}_l(G)) \text{ où } G = \mathtt{D}_{L \setminus \{l\}}(F).$$

Cette définition est valide parce que la formule obtenue ne dépend pas du choix particulier de la lettre l dans l'ensemble L. On peut le prouver en observant que si l_1 et l_2 sont deux lettres différentes et F une formule, on a : $\mathtt{D}_{l_1}(\mathtt{D}_{l_2}(F)) = \mathtt{D}_{l_2}(\mathtt{D}_{l_1}(F))$. Pour le voir, il suffit d'écrire F sous la forme $\overline{l}_1.\overline{l}_2.F_{00} \cup \overline{l}_1.l_2.F_{01} \cup l_1.\overline{l}_2.F_{10} \cup l_1.l_2.F_{11} \cup F_*$, de faire le calcul des deux expressions à partir de cette décomposition et de constater qu'on arrive au même résultat [1].

La méthode de calcul des impliquants premiers de Thayse est basée sur le théorème suivant.

Théorème 2. *Soit F, une formule et soit L, l'ensemble des lettres utilisées par F, alors, la formule $\mathtt{D}_L(F)$ est égale à la liste des impliquants premiers de F.*

La preuve de ce théorème repose sur le lemme ci-dessous.

Lemme 5. *Soit F, une formule utilisant l'ensemble de lettres L_F. Soit L, un ensemble de lettres tel que $L \subseteq L_F$. Soit t un terme utilisant (au moins) toutes les lettres de $L_F \setminus L$. Si t est un impliquant de F, t implique un terme de la formule $\mathtt{D}_L(F)$.*

Nous démontrons le lemme par récurrence sur le nombre d'éléments de l'ensemble L.

1. Mais on peut aussi se passer de cette preuve de validité, en fixant un ordre arbitraire pour le choix des lettres.

Preuve du lemme 5.

1. Si $L = \{\}$, le terme t est un terme canonique de F et il implique évidemment un terme de F, c'est-à-dire de $\mathtt{D}_L(F)$.
2. Si $L = \{l\} \cup L'$, où L' ne contient pas l, et si t est un impliquant qui utilise toutes les lettres de $L_F \setminus L$, nous considérons deux cas.
   (a) Le terme t utilise aussi la lettre l. Dans ce cas, il utilise toutes les lettres de $L_F \setminus L'$ et nous pouvons supposer, par hypothèse de récurrence, qu'il implique un terme de la formule G, où $G = \mathtt{D}_{L'}(F)$. Il implique donc aussi un terme de $\mathtt{D}_L(F) = \mathtt{absorber}(G \cup D_l(G))$.
   (b) Le terme t n'utilise pas la lettre l. Alors, les termes $l.t$ et $\overline{l}.t$ sont des impliquants de F qui utilisent toutes les lettres de $L_F \setminus L'$. Par hypothèse de récurrence, il existe des termes t_0 et t_1 de $\mathtt{D}_{L'}(F)$ qui sont impliqués, respectivement, par $\overline{l}.t$ et $l.t$. Soient t'_0 et t'_1, les termes obtenus en retirant de t_0 et t_1 le littéral portant sur la lettre l qu'ils contiennent éventuellement. Les termes t'_0 et t'_1 sont joignables car leurs ensembles de littéraux forment chacun un sous-ensemble des littéraux de t. Donc, t implique la jointure de t'_0 et t'_1.

      Pour terminer la preuve, il suffit de montrer que $\mathtt{D}_L(F)$ contient un terme qui est impliqué par la jointure de t'_0 et t'_1. On a que $\mathtt{D}_L(F) = \mathtt{absorber}(G \cup (G_0 \cup G_*)\,\hat{.}\,(G_1 \cup G_*))$ où $G = \mathtt{D}_{L'}(F)$ et $\overline{l}.G_0 \cup l.G_1 \cup G_*$ est la décomposition de G par rapport à la lettre l. Il est clair que la jointure de t'_0 et t'_1 appartient à $(G_0 \cup G_*)\,\hat{.}\,(G_1 \cup G_*)$ puisque t'_0 appartient à $G_0 \cup G_*$ et t'_1 appartient à $G_1 \cup G_*$. Donc, par définition de l'opération `absorber`, $\mathtt{D}_L(F)$ contient un terme qui est impliqué par la jointure de t'_0 et t'_1, donc par t. □

Nous pouvons maintenant prouver le théorème principal.

Preuve du théorème 2. En appliquant le lemme 5 au cas où $L = L_F$, nous obtenons directement que tout impliquant de F implique un terme de $\mathtt{D}_L(F)$. Donc, tout impliquant premier implique un terme de $\mathtt{D}_L(F)$. Comme $\mathtt{D}_L(F)$ est logiquement équivalente à F, elle ne peut contenir de terme impliqué par un impliquant premier qui serait différent de cet impliquant premier. Donc, $\mathtt{D}_L(F)$ contient tous les impliquants premiers de F. Par ailleurs, $\mathtt{D}_L(F)$ ne peut contenir aucun autre impliquant de F puisque ceux-ci en ont été retirés par l'opération `absorber`. □

Algorithme de calcul des impliquants premiers

Il est immédiat d'écrire un algorithme itératif de calcul des impliquants premiers d'une formule booléenne en appliquant le théorème ci-dessus. Pour rendre l'algorithme plus efficace, on peut toutefois observer l'identité suivante : quelle que soit la formule $F = \overline{l}.F_0 \cup l.F_1 \cup F_*$, on a :

$$\texttt{absorber}(F \cup \texttt{D}_l(F)) = \texttt{absorber}(F \cup F_0 \mathbin{\hat{.}} F_1).$$

En effet, l'expression $F \cup \texttt{D}_l(F)$ peut s'écrire

$$F \cup (F_0 \cup F_*) \mathbin{\hat{.}} (F_1 \cup F_*)$$

qu'on développe aisément en

$$\overline{l}.F_0 \cup l.F_1 \cup F_* \cup F_0 \mathbin{\hat{.}} F_1 \cup F_0 \mathbin{\hat{.}} F_* \cup F_* \mathbin{\hat{.}} F_1 \cup F_* \mathbin{\hat{.}} F_*.$$

Il est clair que tous les termes des formules $F_0 \mathbin{\hat{.}} F_*$, $F_* \mathbin{\hat{.}} F_1$ et $F_* \mathbin{\hat{.}} F_*$ sont impliqués par au moins un terme de F_*. Donc, les termes de ces trois formules disparaissent si on applique l'opération `absorber` à la formule $F \cup \texttt{D}_l(F)$, sauf s'ils appartiennent à F_*. D'où le résultat annoncé.

Nous présentons maintenant l'algorithme itératif. Il utilise une variable R contenant la formule "courante" et une variable L contenant un ensemble de lettres. Soit F, la formule dont on veut calculer les impliquants premiers. L'algorithme est comme suit.

Initialisation On initialise R à F et L à l'ensemble des lettres de F.

Invariant À une étape quelconque de l'algorithme, on a déjà calculé la dérivée de F pour l'ensemble de lettres L' formé des lettres de F qui ne sont pas dans L. On a : $R = \texttt{D}_{L'}(F)$.

Itération On prélève une lettre l dans L. Puis, on décompose R sous la forme $\overline{l}.R_0 \cup l.R_1 \cup R_*$ et on calcule $R := \texttt{absorber}(R \cup R_0 \mathbin{\hat{.}} R_1)$.

Clôture L'algorithme prend fin quand L est vide.

Exemple Nous appliquons l'algorithme à la formule $F = \overline{a}\overline{b}\overline{c}\overline{d}+\overline{a}\overline{b}c\overline{d}+\overline{a}b\overline{c}d + ab\overline{c}d + \overline{a}\overline{b}cd + a\overline{b}cd + \overline{a}bcd + abcd$, en choisissant successivement dans L, les lettres a, b, c, d.

`Première étape` : $l = a$

$$\begin{aligned} R &= \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + \overline{a}b\overline{c}d + ab\overline{c}d + \overline{a}\overline{b}cd + a\overline{b}cd + \overline{a}bcd + abcd \\ R_0 &= \overline{b}\overline{c}\overline{d} + \overline{b}c\overline{d} + b\overline{c}d + \overline{b}cd + bcd \\ R_1 &= b\overline{c}d + \overline{b}cd + bcd \\ R_0 \widehat{.} R_1 &= b\overline{c}d + \overline{b}cd + bcd \\ \mathtt{D}_{\{a\}}(F) &= \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + b\overline{c}d + \overline{b}cd + bcd \end{aligned}$$

`Seconde étape` : $l = b$

$$\begin{aligned} R &= \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + b\overline{c}d + \overline{b}cd + bcd \\ R_0 &= \overline{a}\,\overline{c}\overline{d} + \overline{a}c\overline{d} + cd \\ R_1 &= \overline{c}d + cd \\ R_0 \widehat{.} R_1 &= cd \\ \mathtt{D}_{\{a,b\}}(F) &= \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + b\overline{c}d + cd \end{aligned}$$

`Troisième étape` : $l = c$

$$\begin{aligned} R &= \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + b\overline{c}d + cd \\ R_0 &= \overline{a}\overline{b}\,\overline{d} + bd \\ R_1 &= \overline{a}\overline{b}\,\overline{d} + d \\ R_0 \widehat{.} R_1 &= \overline{a}\overline{b}\,\overline{d} + bd \\ \mathtt{D}_{\{a,b,c\}}(F) &= \overline{a}\overline{b}\,\overline{d} + bd + cd \end{aligned}$$

`Quatrième étape` : $l = d$

$$\begin{aligned} R &= \overline{a}\overline{b}\,\overline{d} + bd + cd \\ R_0 &= \overline{a}\overline{b} \\ R_1 &= b + c \\ R_0 \widehat{.} R_1 &= \overline{a}\overline{b}c \\ \mathtt{D}_{\{a,b,c,d\}}(F) &= \overline{a}\overline{b}\,\overline{d} + \overline{a}\overline{b}c + bd + cd \end{aligned}$$

La liste des impliquants premiers de la formule F est donc :

$$\overline{a}\overline{b}\,\overline{d} + \overline{a}\overline{b}c + bd + cd.$$

1.4 Comparaison des méthodes

Nous allons maintenant comparer les méthodes présentées dans la section précédente. Nous cherchons à identifier les situations dans lesquelles telle méthode est préférable à telle autre et pour quelles raisons, ainsi que leurs points communs. Nous comparons aussi expérimentalement l'efficacité des méthodes de McCluskey, Coudert et Thayse en nous basant sur une implémentation de chacune d'entre elles au moyen d'une structure de données unique, que nous décrivons.

1.4.1 Premiers éléments de comparaison

Les méthodes proposées par Quine et Karnaugh ont été conçues pour être utilisées sur papier, à la main. Dans cette optique, certains aspects algorithmiques nécessaires à préciser pour une implémentation sur ordinateur, ont été laissés à l'initiative de la personne appliquant la méthode.

La méthode de McCluskey précise certains aspects algorithmiques de la première méthode de Quine mais est encore décrite, dans [McC56], en vue d'une utilisation à la main : les deux variantes que nous en avons données ne sont pas expliquées en tant qu'algorithmes précis mais seulement en tant que "manières de procéder". On peut supposer que, dans l'esprit de l'auteur, le fait de repérer les termes à fusionner ou les combinaisons redondantes de tels termes, pouvait se faire "d'un coup d'œil". Cela se justifie seulement si les exemples à traiter sont de petite taille, de l'ordre de quatre ou cinq variables et d'une vingtaine de termes (taille des exemples proposés dans [McC56]). Nous expliquons plus loin comment nous avons précisé ces aspects dans notre implémentation et ce que cela implique pour l'efficacité des deux variantes de la méthode.

La méthode de Thayse peut être vue comme une spécialisation de la seconde méthode de Quine, qui effectue les opérations de consensus et de retrait des termes qui en impliquent d'autres, dans un ordre plus systématique. Néanmoins, elle ne précise pas comment calculer efficacement l'expression $\texttt{absorber}(R \cup R_0 \widehat{.} R_1)$, sans doute aussi parce que ce calcul peut être fait d'un coup d'œil sur les exemples de petite taille.

En contraste, la méthode de Coudert utilise une approche "diviser pour régner" qui est complètement systématique mais désagréable à appliquer à la main car il faut d'abord découper la formule à traiter en de nombreuses sous-formules jusqu'aux cas de base constitués par les formules 1 et 0, puis construire le résultat final en "remontant dans l'arbre des appels", ce qui est fastidieux.

1.4.2 Méthodes ascendantes et descendantes

On peut classer les méthodes vues en méthodes ascendantes et descendantes. On dit qu'une méthode algorithmique est ascendante si elle construit d'abord les solutions des sous-problèmes les plus simples, puis les étend progressivement jusqu'à obtenir la solution du problème qu'on a en vue. Une méthode est descendante si elle part du problème à résoudre, le décompose en sous-problèmes, résout ceux-ci et remonte progressivement au problème de départ en construisant sa solution. Les méthodes ascendantes sont plus agréables à utiliser à la main car on ne doit pas garder trace de tout ce qui a été fait mais seulement des derniers résultats calculés. Selon cette classification, les méthodes de McCluskey et Thayse sont ascendantes, alors que la méthode de Coudert est descendante.

Une caractéristique commune aux méthodes ascendantes, quand elles sont applicables, est qu'elles ne calculent qu'une fois chacun des sous-problèmes calculés mais qu'elles peuvent en calculer certains qui ne sont pas réellement nécessaires, alors que les méthodes descendantes ne calculent que des sous-problèmes nécessaires, mais elles les calculent parfois plusieurs fois. Les trois méthodes diffèrent néanmoins par la nature des sous-problèmes à résoudre.

Une force essentielle de la méthode de Coudert est qu'elle calcule directement les ensembles d'impliquants premiers, de façon exacte, sans qu'il soit nécessaire d'éliminer les impliquants non premiers comme dans les autres méthodes. Que cela soit possible est en quelque sorte "un coup de chance", propre au problème lui-même. Il n'est apparemment pas possible, avec une méthode ascendante, de découper le problème en sous-problèmes dont la solution est directement formée d'impliquants premiers. Il faut calculer tout (McCluskey) ou partie (Thayse) des impliquants, et éliminer plus tard les impliquants non premiers. Cette nécessité est la principale source de complication et une importante cause d'inefficacité pour toutes les méthodes ascendantes présentées dans ce chapitre.

1.4.3 Passage obligé (ou non) par des formules en forme canonique

À l'exception notable de la méthode de Thayse [2], toutes les méthodes que nous avons présentées présupposent que la formule F, dont on veut calculer la liste des impliquants premiers, est mise sous forme canonique.

2. Et de la seconde méthode de Quine, qu'elle systématise.

Le fait d'être applicable directement à une formule quelconque (sous forme disjonctive, bien sûr) rend la méthode de Thayse attractive pour une utilisation "à la main", vu le caractère fastidieux de la mise sous forme canonique d'une formule sous forme disjonctive quelconque.

C'est particulièrement le cas si la formule de départ utilise beaucoup de lettres et des termes de petite taille. Ainsi, notons que si les termes sont de taille inférieure ou égale à deux, l'opération de consensus produit un nouveau terme de taille inférieure ou égale à deux. L'algorithme de Thayse appliqué à une formule ne contenant que des termes de cette sorte sera donc particulièrement simple à appliquer et efficace, alors même que la mise sous forme canonique de la formule sera extrêmement fastidieuse car conduisant à une formule de taille exponentielle dans le nombre de variables.

Si les termes sont de plus grande taille, l'opération de consensus peut produire des termes plus grands encore et l'application de la méthode de Thayse peut créer beaucoup plus de termes que le nombre de termes canoniques de la formule. Il est alors parfois préférable d'appliquer la méthode de Thayse à la formule canonique qu'à la formule de départ car celle-ci peut contenir des termes qui "se recouvrent" et, du coup, les mêmes impliquants peuvent être calculés plusieurs fois, ce qui n'est pas le cas si l'on part d'une formule sous forme canonique. Nous examinons plus bas diverses situations où l'application de la méthode de Thayse à des formules non canoniques est préférable.

1.4.4 Implémentation des différentes méthodes

Nous décrivons maintenant le mode de représentation des formules que nous avons choisi afin d'implémenter concrètement, sur ordinateur, les principales méthodes présentées à la section 1.3. Nous donnons ensuite des détails sur l'implémentation des algorithmes lorsque leur réalisation ne découle pas directement de la présentation faite à la section 1.3.

Nous représentons les termes par des entiers et les formules par des tableaux triés d'entiers.

À chaque lettre l, nous associons l'entier naturel $\texttt{rang}(l)$ qui est égal à la position de l, dans l'alphabet, moins 1. Donc, $\texttt{rang}(\text{'}a\text{'}) = 0$ et $\texttt{rang}(\text{'}p\text{'}) = 15$. Nous nous limitons à des formules utilisant les 16 premières lettres de l'alphabet, afin de pouvoir représenter n'importe quel terme par un entier (non signé) sur 32 bits. (Cette restriction peut bien sûr être levée en utilisant de plus grands entiers ou d'autres formes de "chaines de bits".)

Soit t, un terme, nous le représentons par l'entier naturel $\texttt{repr}(t)$ dont

le bit de poids $16 + m$ vaut 1, si $m = \texttt{rang}(l)$ et t contient le littéral l ($m < 16$) ; ce bit vaut 0 sinon. De plus, si t contient le littéral $\overline{l}$, le bit de poids m de $\texttt{repr}(t)$ vaut 1 ; il vaut 0, sinon.

La formule $t_1 + \cdots + t_n$ est représentée par un tableau d'entiers, trié selon l'ordre naturel des entiers, et contenant les valeurs $\texttt{repr}(t_1), \cdots, \texttt{repr}(t_n)$ (voir figure 1.1).

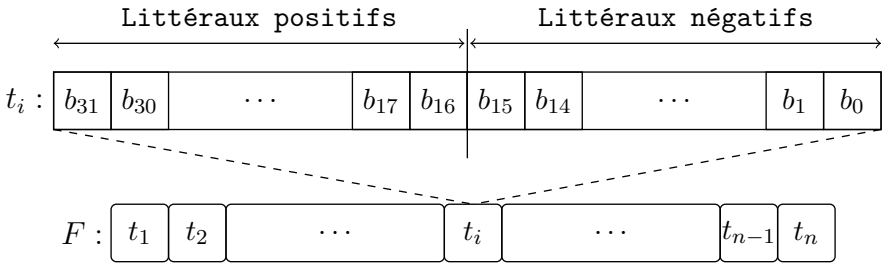


Figure 1.1 – Représentation interne d'une formule booléenne F

En particulier, la formule 0 est représentée par le tableau vide et la formule 1 (contenant le seul terme 1) est représentée par un tableau d'un seul élément, égal à 0.

Notre façon de représenter les formules est compacte et efficace pour calculer les opérations de base (union, différence, intersection, ...) sur les formules considérées comme des ensembles de termes. Pour manipuler les termes, nous utilisons les opérations logiques sur les entiers considérés comme des suites de bits (disponibles en Java, notre langage d'implémentation).

Il peut nous être fait remarquer qu'une représentation sous forme de BDDs [Bry86, Min93a, Min93b, Min01] serait plus compacte et efficace, en particulier pour implémenter l'algorithme de Coudert. Cela n'est pas vrai dans tous les cas car toutes les formules n'ont pas une représentation compacte sous forme de BDD. (Dans notre représentation, la taille des formules ne dépend pas de "l'ordre des variables".) Mais l'intérêt principal de notre représentation est de convenir à l'implémentation de toutes les méthodes étudiées dans ce chapitre, ce qui n'est pas le cas d'une représentation sous forme de BDDs. Cela permet donc de comparer plus directement ces méthodes. Par ailleurs, il n'est pas essentiel à la méthode de Coudert d'être implémentée avec des BDDs. Conceptuellement, elle manipule des ensembles de termes et notre représentation est efficace pour représenter de tels ensembles. Elle permet, en particulier, de réaliser tous les algorithmes manipulant les formules selon l'approche "diviser pour régner".

Nous avons implémenté les différentes méthodes avec notre mode de

représentation des formules. Nous expliquons maintenant comment sont réalisées certaines phases des algorithmes qui n'ont pas été précisées dans la section 1.3, décrivant les méthodes, parce que ces précisions n'étaient pas nécessaires pour une utilisation des méthodes, "à la main", sur des exemples simples.

Dans la variante 1 de la méthode de McCluskey, nous devons rechercher dans chaque liste $li(x+1)$, les termes t' qui sont fusionnables à chaque terme t de la liste $li(x)$. Pour ne créer qu'une fois les termes fusionnés, nous recherchons uniquement les termes dont la représentation est supérieure à celle de tous les termes complets compatibles avec t. Par exemple, si t est le terme $\overline{a}b\overline{d}$ et qu'on utilise quatre lettres seulement, le plus grand terme complet compatible avec $\overline{a}b\overline{d}$ est $\overline{a}bc\overline{d}$, représenté en binaire par 0110 1001 [3]. Les termes t', fusionnables avec t, sont $ab\overline{d}$ et $\overline{a}bd$ qui ont, respectivement, pour représentation binaire 0011 1000 et 1010 0001. Seul $\overline{a}bd$ a une représentation supérieure à celle de $ab\overline{c}\overline{d}$ et il sera donc seul recherché (par recherche dichotomique). Sa représentation est facilement calculée à partir de celle de $\overline{a}b\overline{d}$, soit 0010 1001, en inversant les bits correspondant à la lettre d. Chaque fois qu'une fusion est obtenue, dans une liste $li'(x)$, il faut encore marquer les termes des listes $li(x-1)$ et $li(x+1)$ susceptibles de produire le même terme t', par fusion. On peut déterminer, de manière semblable, les représentations binaires de ces termes, à partir de celle de t', et rechercher la présence de ces termes par recherche dichotomique dans les listes $li(x-1)$ et $li(x+1)$.

La variante 2 de la méthode de McCluskey ne nécessite pas d'explications particulières : les listes $li(x)$ et $li(x+1)$, sont aisément réparties en sous-listes de termes contenant la même lettre, en utilisant la représentation binaire des termes ; ensuite, les sous-listes sont parcourues en parallèle en utilisant l'ordre naturel sur les entiers ; finalement les sous-listes résultantes sont fusionnées en parallèle.

La méthode de Coudert est facilement implémentée en utilisant notre représentation des formules. Les opérations d'union, de différence, et d'intersection sont efficacement implémentées grâce à l'utilisation de tableaux triés. Nous avons aussi besoin d'une opération qui décompose une formule canonique F en deux formules F_0 et F_1 telles que $F = \overline{l}.F_0 \cup l.F_1$, ainsi qu'une opération qui construit une formule non canonique F, de la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$, à partir des formules F_0, F_1 et F_* (où aucune des formules n'utilise la lettre l). Ces deux opérations peuvent être réalisées en temps linéaire sur le nombre de termes de F.

Concernant la méthode de Thayse, le problème se pose d'implémenter efficacement les opérations `absorber`(F) et $F \hat{} G$. Une implémentation

3. Nous omettons les bits correspondant aux lettres non utilisées.

directe de ces opérations est, respectivement, quadratique dans la taille de F ou proportionnelle au produit des tailles des formules F et G. Cette complexité est acceptable pour des formules de petite taille. Pour des formules de grande taille, nous avons implémenté des algorithmes qui utilisent la même approche "diviser pour régner" que l'algorithme de Coudert. Ces algorithmes sont décrits dans l'annexe A.

1.4.5 Comparaison expérimentale des méthodes

Nous comparons maintenant les performances des différentes méthodes en identifiant différents cas de figure mettant en avant leurs points forts et leurs points faibles.

Formules sous forme canonique

Nous montrons d'abord les résultats de mesures effectuées sur des formules de différentes tailles, "choisies au hasard" et utilisant 16 lettres. Les méthodes de McCluskey et Coudert n'étant applicables qu'à des formules canoniques, nous nous limitons, dans cette expérimentation, à de telles formules. La taille d'une formule canonique utilisant 16 lettres est de 65.536 termes au plus. Nous divisons l'intervalle 1..65.536 en 16 intervalles de 4.096 valeurs consécutives. Pour chaque intervalle nous choisissons 100 formules "au hasard", en choisissant d'abord une taille n, puis une formule canonique composée de n termes de 16 lettres, choisis aléatoirement mais distincts. Nous appliquons à ces formules quatre méthodes de calcul des impliquants premiers : les deux variantes de la méthode de McCluskey ($\texttt{McC}_1$, $\texttt{McC}_2$), la méthode de Coudert (`Cdrt`), et une variante de la méthode de Thayse utilisant une implémentation du type "diviser pour régner" (désignée par $\texttt{Tha}_2$). (Nous avons aussi réalisé une autre variante de la méthode de Thayse utilisant une implémentation simple (quadratique) des méthodes `absorber` et $\widehat{\ }$ (désignée par $\texttt{Tha}_1$). Mais nous ne l'utilisons pas dans cette première expérimentation car elle prend trop de temps pour de grandes formules.) Enfin, nous mesurons les temps d'exécution des différents algorithmes et nous faisons une moyenne sur l'ensemble des formules choisies dans chaque intervalle. Nous calculons aussi la taille moyenne des formules et le nombre moyen d'impliquants premiers obtenus. Les résultats de ces mesures sont présentés à la table 1.6.

On présente également, à la figure 1.2, quatre courbes correspondant aux moyennes des temps d'exécution des différentes méthodes divisés par le nombre de termes des formules en entrée, ainsi qu'une courbe ($\#IP/n$) qui fournit le rapport entre le nombre d'impliquants premiers

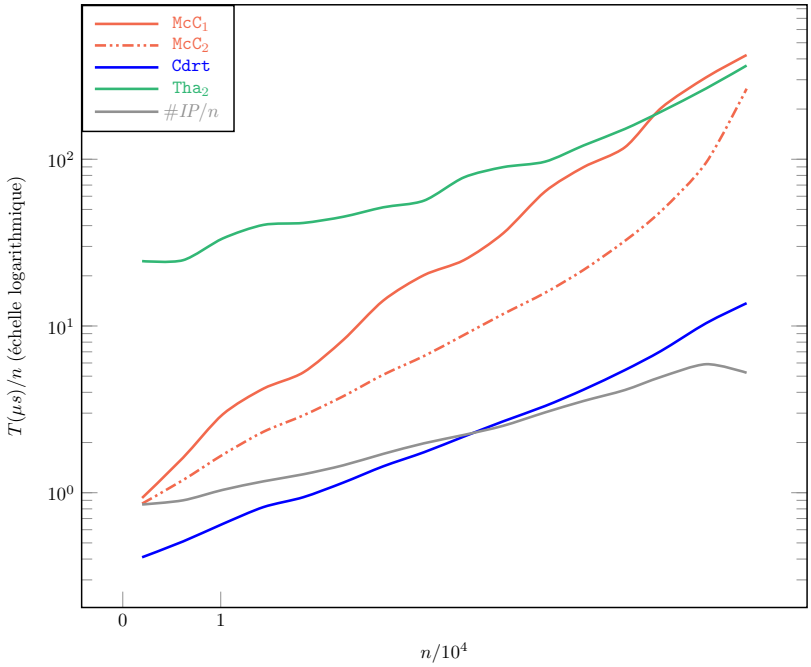


Figure 1.2 – Temps d'exécution en fonction de la taille des formules (Test 1)

et le nombre de termes des formules. Une échelle logarithmique est utilisée. On constate que la méthode de Coudert est la plus efficace. Les deux versions de la méthode de McCluskey ont une efficacité comparable mais la version qui ne calcule qu'une fois chaque impliquant premier est moins efficace, probablement parce qu'elle fait beaucoup de recherches infructueuses de termes à fusionner. La méthode de Thayse est moins efficace que les deux variantes de la méthode de McCluskey sur ces cas de tests ; néanmoins, elle présente un tracé semblable à celui de la méthode de Coudert. La différence d'efficacité peut s'expliquer par le fait que les opérations `absorber` et $\widehat{\cdot}$ sont chacune appliquées 16 fois. En supposant que ces opérations ont une complexité comparable à l'unique opération de calcul des impliquants premiers de Coudert, cela permet déjà d'envisager un temps d'exécution plus grand d'un facteur 32 et c'est à peu près ce qu'on observe expérimentalement. Nous constatons enfin que la courbe $\#IP/n$ croît moins vite que les courbes `Cdrt` et $\texttt{Tha}_2$, indiquant que ces méthodes, et a fortiori les deux autres, ont un temps d'exécution qui croît plus vite que proportionnellement au nombre d'impliquants premiers de la formule.

Les valeurs obtenues dans ce premier ensemble de tests montre que l'efficacité des différentes méthodes, relative à taille des formules, diminue

n	$\#IP$	McC$_1$		McC$_2$		Cdrt		Tha$_2$	
		$T(\mu s)$	T/n	$T(\mu s)$	T/n	$T(\mu s)$	T/n	$T(\mu s)$	T/n
1.972	1.670	1.825	0	1.697	0	816	0	48.380	24
6.147	5.536	9.989	1	7.329	1	3.146	0	152.340	24
10.206	10.653	30.129	2	17.217	1	6.617	0	341.494	33
14.359	16.751	60.394	4	33.297	2	11.771	0	579.466	40
18.435	23.691	97.080	5	53.722	2	17.416	0	765.278	41
22.510	32.952	185.626	8	85.230	3	25.861	1	1.017.620	45
26.588	45.519	378.294	14	135.589	5	38.401	1	1.370.354	51
30.760	60.876	621.466	20	203.468	6	53.829	1	1.739.134	56
34.807	77.240	863.834	24	307.140	8	75.588	2	2.707.268	77
38.838	98.046	1.398.425	36	457.196	11	104.240	2	3.484.356	89
43.101	130.638	2.759.283	64	683.130	15	142.605	3	4.167.385	96
47.040	167.119	4.218.861	89	1.022.834	21	195.291	4	5.708.291	121
51.260	211.883	6.031.131	117	1.659.867	32	279.083	5	7.782.243	151
54.963	272.054	11.166.756	203	2.686.850	48	389.908	7	10.580.943	192
59.558	351.466	18.480.373	310	5.695.781	95	619.797	10	15.836.939	265
63.687	333.527	26.849.757	421	16.891.302	265	875.217	13	23.275.538	365

Table 1.6 – Temps d'exécution des méthodes (Test 1)

fortement quand cette taille augmente. Mais nous ne pouvons en déduire, par exemple, que la complexité serait exponentielle par rapport à la taille des formules puisque cette taille est, ici, bornée par 65.536. Nous pouvons, par contre, émettre l'hypothèse que la complexité des différentes méthodes augmente lorsque la taille des formules se rapproche du maximum possible pour le nombre de lettres qu'elles utilisent. Pour mettre à l'épreuve cette conjecture, nous réalisons deux autres séries de tests en choisissant, d'une part, une classe de formules "faciles" et, d'autre part, une classe de formules "difficiles". Nous faisons varier le nombre m de lettres des formules de 4 à 16 et nous générons "aléatoirement" des formules de taille égale soit à $2^m - 2^{m-4}$ (formules "difficiles"), soit à 2^{m-4} (formules "faciles"). Nous mesurons à nouveau la moyenne des temps d'exécution et le nombre moyen d'impliquants premiers calculés. Les résultats sont présentés aux figures 1.3 et 1.4. Nous constatons expérimentalement que, dans les deux cas, les courbes reprenant les temps moyens des différentes méthodes présentent des tracés quasi parallèles, signifiant que les méthodes ont pratiquement la même complexité théorique, mais avec des "constantes de proportionnalité" différentes. De plus, cette complexité est du type $O(mn)$ pour les formules "faciles" et du type $O(n^{\log_2(3)})$ pour les formules "difficiles". Pour le montrer, nous avons ajouté la courbe de la fonction mn aux courbes du test "facile" et la courbe de la fonction $n^{\log_2(3)}$ aux courbes du test "difficile". Nous observons également que la courbe qui donne le nombre d'impliquants premiers croît également, à peu près, comme les temps

d'exécution. Donc, nous pouvons conclure "expérimentalement" que les temps d'exécution de toutes les méthodes croît proportionnellement aux nombres d'impliquants premiers, *pour les formules de même niveau de difficulté* (en fait, un peu plus rapidement). Nous pouvons conclure aussi que le nombre d'impliquants premiers est en $O(mn)$ pour les formules "faciles" et en $O(n^{\log_2(3)})$ pour les formules "difficiles".

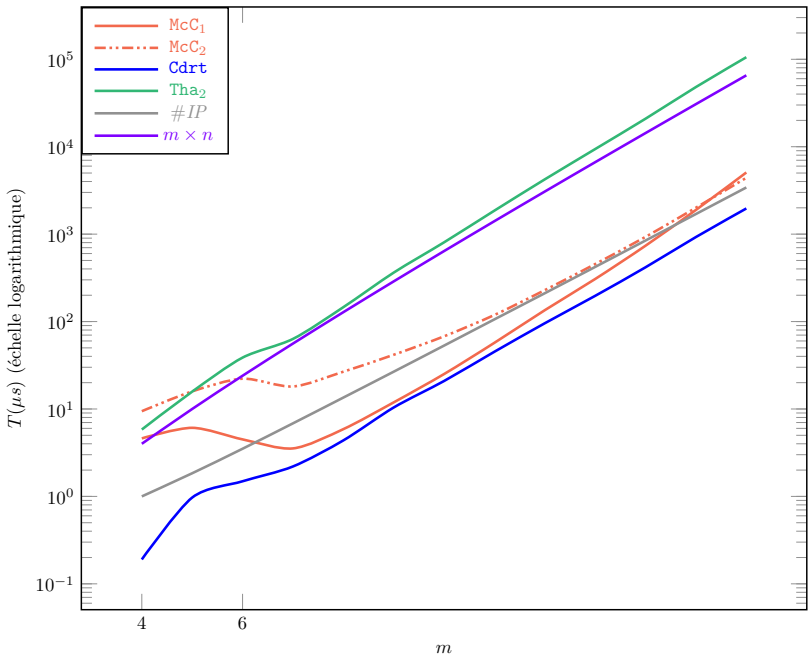


Figure 1.3 – Temps d'exécution pour les formules "faciles"

Finalement, nous ajoutons à cette analyse expérimentale des considérations plus théoriques cohérentes avec nos observations précédentes. Attachons-nous à estimer la complexité théorique de l'algorithme de Coudert (voir la section 1.3.4). Notons $T(n, m)$, le temps d'exécution moyen (estimé) de l'algorithme pour une formule F de n termes utilisant m lettres. Nous estimons que, pour $m > 0$:

$$T(n, m) \leq kn + T(n_0, m-1) + T(n_1, m-1) + T(n_*, m-1)$$

où k est une constante de proportionnalité "assez grande" et n_0, n_1, n_* sont les nombres de termes des formules F_0, F_1, $F_1 \cap F_0$ (voir l'algorithme). En moyenne, nous pouvons estimer que

$$n_0 = n_1 = \frac{n}{2}.$$

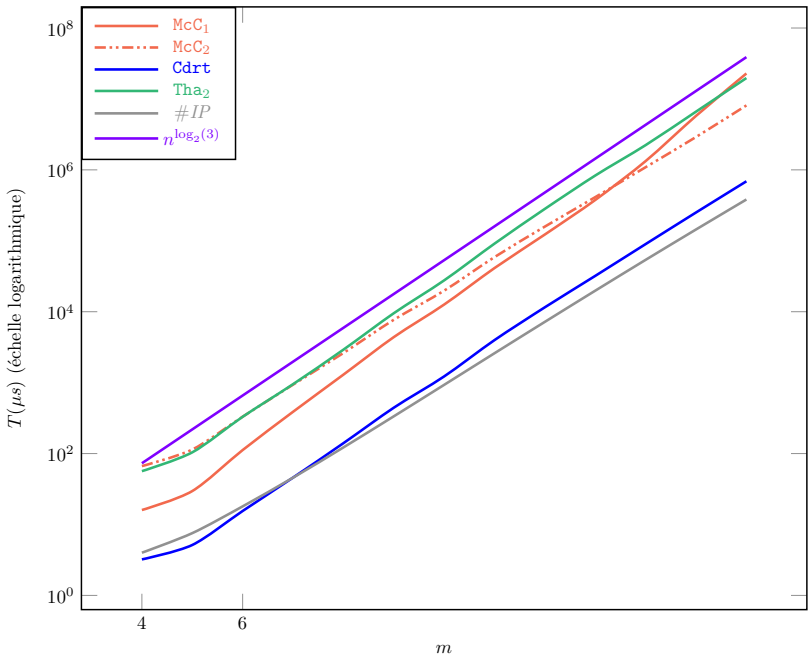


Figure 1.4 – Temps d'exécution pour les formules "difficiles"

Si la formule est "facile", la formule $F_1 \cap F_0$ contient peu d'éléments, voire aucun. Alors, nous pouvons estimer que $T(n_*, m-1)$ est négligeable. L'estimation de $T(n,m)$ devient :

$$T(n,m) \;\leq\; k\ n \;+\; 2\ T(\frac{n}{2}, m-1).$$

Le calcul nous donne facilement :

$$T(n,m) \;\leq\; k\ m\ n,$$

c'est-à-dire $T(n,m) = O(mn)$. Si, par contre, la formule est "difficile", la formule $F_1 \cap F_0$ contient presque tous les termes de F_0 et de F_1. On simplifie l'estimation de $T(n,m)$ en

$$T(n,m) \;\leq\; k\ n \;+\; 3\ T(\frac{n}{2}, m-1).$$

d'où l'on tire, par le calcul :

$$T(n,m) \;\leq\; k\ n\ \frac{(\frac{3}{2})^{m+1}-1}{\frac{3}{2}-1} \;\leq\; 3\,k\,n\,\left(\frac{3}{2}\right)^m.$$

Comme la formule est "difficile", on a : $n \approx 2^m$, soit $m \approx \log_2(n)$. Donc :

$$3\,k\,n\,\left(\frac{3}{2}\right)^m \approx 3\,k\,2^m\,\left(\frac{3}{2}\right)^m \;=\; 3\,k\,3^m \;\approx\; 3\ k\ 3^{\log_2(n)} = 3\ k\ n^{\log_2(3)}.$$

On en conclut bien que $T(n,m) = O(n^{\log_2(3)})$. On peut noter finalement que $1,5 \approx \log_2(3) < 1,6$. Donc, la complexité de l'algorithme, pour les formules "difficiles" est en $O(n^{1,6})$.

Formules sous forme disjonctive générale

Nous donnons ensuite un test d'efficacité qui met particulièrement en lumière l'intérêt d'utiliser la méthode de Thayse pour des formules non mises sous forme canonique. Nous effectuons des mesures portant sur les 15 formules $a+b$, $a+b+c$, $\ldots$, $a+b+c+\cdots+p$. Nous appliquons à ces formules les cinq algorithmes spécifiés plus haut, mais nous ne mettons pas les formules sous forme canonique avant de leur appliquer les deux variantes de la méthode de Thayse. Les temps d'exécution obtenus sont présentés dans la table 1.7 où la colonne n_c fournit le nombre de termes de la formule mise sous forme canonique et la colonne T_c, le temps d'exécution nécessaire à la mise sous forme canonique. Ce temps doit être ajouté aux temps obtenus pour les méthodes de McCluskey et de Coudert mais il est, en fait, marginal par rapport au temps d'exécution de ces méthodes.

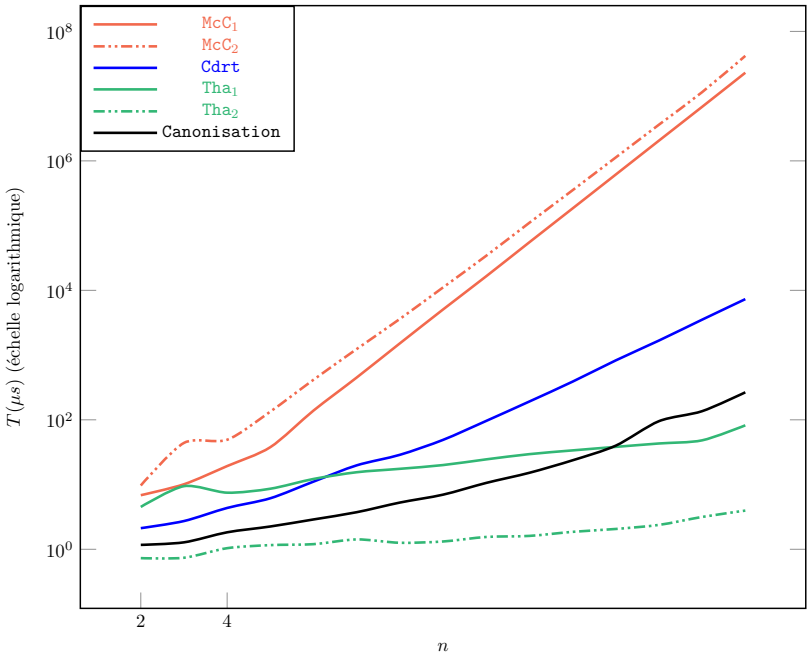


Figure 1.5 – Temps d'exécution en fonction de la taille des formules (Test 2)

Nous proposons également, à la figure 1.5, six courbes représentant les

n	n_c	$T_c(\mu s)$	$T_{\texttt{McC}_1}(\mu s)$	$T_{\texttt{McC}_2}(\mu s)$	$T_{\texttt{Cdrt}}(\mu s)$	$T_{\texttt{Tha}_1}(\mu s)$	$T_{\texttt{Tha}_2}(\mu s)$
2	3	1	5	8	0	4	0
3	7	1	8	42	1	9	0
4	15	1	17	47	2	7	1
5	31	2	35	132	3	8	1
6	63	2	135	412	8	12	1
7	127	3	446	1.236	16	15	1
8	255	5	1.512	3.614	23	17	1
9	511	6	5.058	11.088	42	20	1
10	1.023	10	16.498	34.212	85	24	1
11	2.047	15	55.368	110.799	176	29	1
12	4.095	23	184.318	352.100	365	33	1
13	8.191	39	615.274	1.130.018	787	38	2
14	16.383	95	2.055.365	3.623.830	1.582	43	2
15	32.767	135	6.843.785	11.532.189	3.389	48	3
16	65.535	265	23.007.813	41.897.419	7.054	82	3

Table 1.7 – Temps d'exécution des méthodes (Test 2)

mêmes résultats. Une échelle logarithmique est utilisée à nouveau. Nous constatons, cette fois, que la méthode de Thayse, y compris sa variante la plus simple, l'emporte haut la main sur les autres méthodes. Le temps d'exécution de la méthode de Coudert évolue proportionnellement à la taille de la formule canonique correspondant à la formule d'entrée, c'est-à-dire exponentiellement par rapport au nombre de lettres. Les deux méthodes de McCluskey donnent les moins bons résultats et se comportent de manières similaires, avec, cette fois, un avantage à la première variante qui ne calcule qu'une fois chaque impliquant. Ces cas de figure sont d'ailleurs les pires pour la méthode de McCluskey car le nombre d'impliquants de ces formules est égal à $3^m - 1$ (où m est le nombre de lettres de la formule). Chacun de ces impliquants étant calculé au moins une fois, le temps d'exécution de la méthode est au minimum proportionnel à 3^m. Le temps d'exécution par rapport à la taille n_c de la formule *canonique* d'entrée est donc au minimum de l'ordre de $n_c^{1,5}$, car $3^m = 2^{m \log_2(3)} \approx n_c^{\log_2(3)} \approx n_c^{1,5}$.

Nous donnons finalement un dernier ensemble de tests dans le but de tracer des limites dans lesquelles la méthode de Thayse est compétitive avec celle de Coudert et/ou préférable à celle de McCluskey. Nous créons des formules utilisant 16 lettres. Pour chaque entier l compris entre 2 et 16, nous générons 100 formules choisies "au hasard" et formées de 20 termes de taille l, utilisant l lettres, choisies parmi les 16 lettres possibles. Nous appliquons à ces formules les méthodes $\texttt{McC}_1$, $\texttt{McC}_2$, $\texttt{Cdrt}$ et $\texttt{Tha}_2$. La méthode $\texttt{Tha}_1$ n'est pas utilisée car elle présente de trop

mauvaises performances pour certaines valeurs de l. Nous calculons les moyennes des temps d'exécution pour ces formules, en fonction de la valeur l. Nous donnons aussi les temps nécessaires pour mettre les formules sous forme canonique et nous ajoutons ces temps à celui des méthodes $\texttt{McC}_1$, $\texttt{McC}_2$ et `Cdrt`, pour rendre plus équitable la comparaison de ces méthodes avec la méthode $\texttt{Tha}_2$. Nous calculons aussi la moyenne des tailles des formules mises sous forme canonique et des nombres d'impliquants premiers de ces formules. Les résultats de ces tests sont donnés dans la table 1.8. Comme précédemment, nous représentons les mêmes résultats sous forme de courbes, à la figure 1.6.

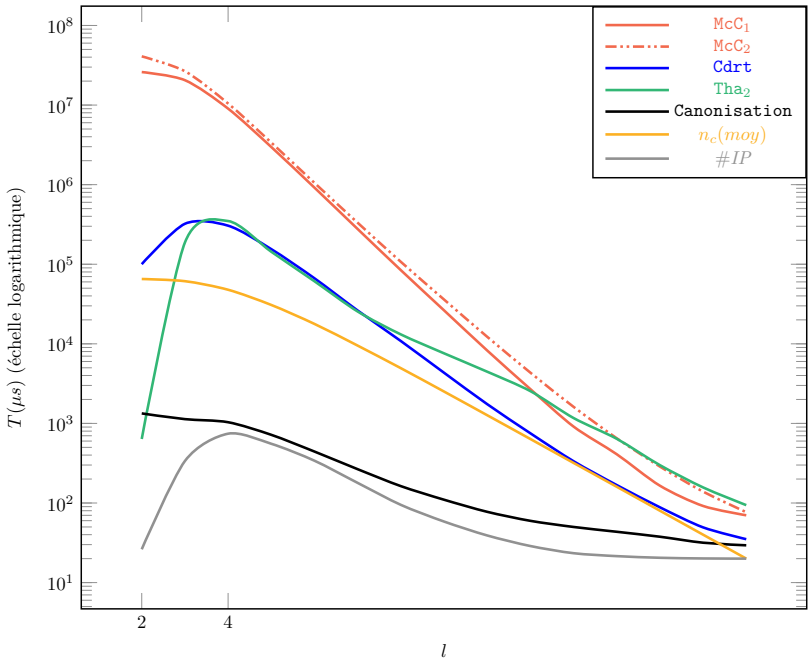


Figure 1.6 – Temps d'exécution en fonction de la taille des formules (Test 3)

Nous constatons que la méthode de Thayse est plus efficace que les autres pour les petites valeurs de l mais ces performances se dégradent ensuite quand la valeur de l augmente. Nous voyons néanmoins que la méthode de Thayse reste toujours compétitive avec les autres méthodes, y-compris celle de Coudert. Ses performances ne sont significativement moins bonnes que celle de la méthode de Coudert que lorsque la taille des formules canoniques devient petite (< 5000), c'est-à-dire quand toutes les méthodes sont quasi instantanées.

Par rapport à la méthode de McCluskey, la méthode de Thayse four-

l	$n_c(moy)$	$\#IP$	$T_c(\mu s)$	$T_{\texttt{McC}_1}(\mu s)$	$T_{\texttt{McC}_2}(\mu s)$	$T_{\texttt{Cdrt}}(\mu s)$	$T_{\texttt{Tha}_2}(\mu s)$
2	65.227	26	1.336	25.980.225	40.917.381	98.798	637
3	61.257	337	1.132	20.534.938	26.597.653	321.298	187.649
4	47.809	743	1.035	9.021.979	10.460.676	304.167	348.845
5	30.798	555	719	2.971.883	3.377.456	154.090	143.556
6	17.792	340	443	916.597	1.052.144	66.152	61.100
7	9.501	179	266	276.851	332.065	25.950	25.487
8	4.945	94	163	84.116	107.103	10.686	13.134
9	2.516	58	110	26.245	35.939	4.224	7.602
10	1.268	39	77	8.168	12.184	1.669	4.465
11	637	28	59	2.639	4.286	693	2.546
12	319	23	50	863	1.566	285	1.169
13	160	21	43	367	608	125	641
14	80	21	37	128	250	51	299
15	40	20	31	60	108	18	157
16	20	20	29	41	47	6	94

Table 1.8 – Temps d'exécution des méthodes (Test 3)

nit, ici, de bien meilleurs résultats ce qui n'était pas le cas pour notre premier groupe de tests. Pour le comprendre, on peut d'abord observer que le nombre d'impliquants premiers des formules est beaucoup plus petit dans ce nouveau test que dans le premier, les termes de ces formules étant déjà pour la plupart des impliquants premiers. La méthode de McCluskey est comparativement plus efficace quand la formule canonique contient un très grand nombre d'impliquants premiers : ceux-ci, étant pour la plupart de grande taille, sont détectés aux premières étapes de l'algorithme. Par contre, si une formule canonique contient un grand nombre de termes mais un petit nombre d'impliquants premiers, ceux-ci sont de petite taille et sont identifiés tardivement. Donc, beaucoup d'impliquants non premiers sont calculés, augmentant le temps d'exécution. Lorsqu'on choisit des formules canoniques d'assez grande taille, "au hasard", la probabilité d'avoir peu d'impliquants premiers est faible. Par contre, si on part d'une formule courte avec des termes courts, on obtiendra aussi de grandes formules canoniques mais des formules avec peu d'impliquants premiers.

Conclusion de l'étude expérimentale

Notre étude expérimentale montre la supériorité de la méthode de Coudert sur les autres méthodes lorsque la formule de départ est sous forme canonique. Pour les formules sous forme disjonctive quelconque, la conclusion est moins claire et la méthode de Thayse est parfois beaucoup plus efficace. La question de l'existence d'algorithmes "polynomiaux" pour calculer la liste des impliquants premiers d'une formule sous forme nor-

male disjonctive a occupé de nombreux auteurs (voir [Str92]) mais, en général, ces auteurs mesurent la complexité des méthodes en fonction de la taille d'une formule sous forme canonique. Nous avons montré expérimentalement que toutes les méthodes que nous avons présentées ont une complexité en $O(n^{1,6})$, ce qui est non seulement polynomial mais même presque optimal, étant donné que le nombre d'impliquants premiers croît presqu'aussi vite.

Il est donc plus correct de poser la question de la complexité d'une méthode de calcul des impliquants premiers pour des formules sous forme normale disjonctive *quelconque*. Nous avons montré que pour la classe des formules formées de n termes réduits à une seule variable, la méthode de Thayse est linéaire alors que les autres méthodes sont exponentielles parce qu'elles s'appliquent à une formule canonique et que la taille de celle-ci est exponentielle sur le nombre de variables. Il n'est donc pas possible de concevoir une méthode polynomiale pour calculer la liste des impliquants premiers d'une formule quelconque, sous forme normale disjonctive. Il est, par ailleurs, montré dans [CM78] que le nombre d'impliquants premiers peut être exponentiel dans le nombre de termes de la formule (sans restriction sur le nombre de variables). Ces auteurs le démontrent en proposant une classe de formules pour lesquelles c'est le cas. Une formule de cette classe est la suivante :

$$\overline{a}\overline{b}\overline{c}d + \overline{a}\overline{b}\overline{c}e + \overline{a}\overline{b}\overline{c}f + \overline{a}\overline{b}cg + \overline{a}\overline{b}ch + \overline{a}\overline{b}ci + \overline{a}bj + \overline{a}bk + \overline{a}bl + am + an + ao.$$

Elle contient seulement 12 termes mais possède 255 impliquants premiers. Notre implémentation de la méthode de Thayse les calcule en 16 millisecondes contre 307 millisecondes pour la méthode de Coudert et environ 10 secondes pour les deux variantes de la méthode de McCluskey.

1.5 Conclusion

Nous avons étudié, dans ce chapitre, quelques méthodes de calcul des impliquants premiers d'une formule booléenne, écrite sous forme normale disjonctive. Ces méthodes sont représentatives de l'ensemble des méthodes proposées dans la littérature. Nous avons présenté les différentes méthodes dans un cadre unifié, en fixant une syntaxe très précise pour les formules, plus stricte que celle proposée d'habitude dans la littérature. Cela nous permet de décrire les méthodes "formellement" au moyen d'un petit nombre d'opérations sur les formules et les termes. Nous avons prouvé rigoureusement la correction des méthodes, hormis les cas où elle était évidente. Nous avons, de plus, proposé une représentation interne des formules, sur ordinateur, qui est en lien direct avec la notation utilisée sur papier, rendant immédiate la traduction des opérations formelles en algorithmes et en programmes. Enfin, nous avons effectivement implémenté sur ordinateur les différentes méthodes étudiées afin de comparer expérimentalement leurs performances pour différentes classes de formules. Nous en avons tiré des conclusions concernant leur efficacité respective et leur complexité.

Pour clore cette étude, nous rappelons certaines constations et conclusions énoncées dans le corps de ce chapitre, qui nous semblent intéressantes ou importantes. Parmi les méthodes étudiées, les plus agréables à utiliser à la main, sur papier, sont les méthodes de McCluskey et de Thayse. Mais celles-ci peuvent être vues comme des améliorations des deux méthodes proposées par Quine. Ces dernières sont, en fait, déjà suffisamment précises pour être utilisées à la main, si on les applique à des formules simples. La méthode de Thayse présente sur celle de McCluskey l'avantage de ne pas calculer tous les impliquants d'une formule donnée. Elle a aussi l'avantage d'être applicable directement à une formule sous forme disjonctive quelconque sans passer par une mise sous forme canonique. Néanmoins, le calcul des opérations utilisées par la méthode de Thayse est plus difficile à organiser systématiquement, ce qui explique son efficacité moindre pour des formules mises sous forme canonique. Parmi les deux variantes étudiées de la méthode de McCluskey, aucune ne l'emporte sur l'autre dans tous les cas. La première variante ne calcule qu'une fois chaque impliquant mais fait plus d'essais infructueux que la seconde qui parcourt des listes de termes en parallèle. Il s'en suit que la première variante l'emporte sur la seconde quand la formule admet un petit nombre d'impliquants premiers et la seconde l'emporte sur la première quand il y a beaucoup d'impliquants premiers. En effet, paradoxalement, peu d'impliquants premiers signifie beaucoup d'impliquants non premiers, donc peu de recherches infructueuses. Dans l'autre

sens, beaucoup d'impliquants premiers signifie relativement peu d'impliquants non premiers. Si l'on néglige le coût de la mise sous forme canonique d'une formule, la méthode qui semble pouvoir être choisie comme maître achat pour une exécution sur ordinateur est celle de Coudert qui exploite la propriété remarquable que les impliquants premiers peuvent être calculés en utilisant une stratégie "diviser pour régner" sur la décomposition des formules canoniques par rapport aux littéraux positifs et négatifs portant sur la même lettre. Cela évite notamment de recourir à des opérations de simplification, ou de détection d'impliquants non premiers, comme dans les autres méthodes. Par contre, cette méthode, descendante, n'est pas pratique pour une utilisation sur papier. Notons aussi que, si la méthode de Coudert a été implémentée originellement avec une structure de données de type BDDs, il est possible de le faire aussi bien en utilisant notre représentation simple au moyen de tableaux. La représentation sous forme de BDDs (ou plus exactement de ZBDDs [Min93b]) est normalement plus efficace car les opérations de composition/décomposition d'une formule F, sous la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$ se fait en temps constant avec ces structures. Néanmoins d'autres opérations sur les formules comme la différence et l'intersection sont aussi concernées. Il s'en suit qu'une différence théorique de complexité entre les deux représentations n'est pas absolument garantie.

Pour terminer sur une note plus personnelle, nous suggérons que des travaux pourraient être faits pour déterminer des classes "intéressantes" de formules sous forme normale disjonctive pour lesquelles telles ou telles méthodes sont les plus efficaces et, peut-être, de nouvelles méthodes particulièrement adaptées à ces classes de formules.

Chapitre 2

Une structure de données pour manipuler de grands ensembles de termes égaux

2.1 Introduction

Le problème de simplifier une expression dans un cadre mathématique donné [1] ne peut, en général, pas être résolu de manière satisfaisante sans passer en revue un grand nombre d'expressions intermédiaires. Il est très souvent nécessaire de considérer des expressions plus grandes que l'expression de départ, mais qui se simplifieront mieux. L'énumération explicite de toutes ces expressions, une à une, conduit à des algorithmes trop inefficaces pour la pratique. Cette constatation nous a suggéré de chercher un moyen de manipuler non pas une seule, mais tout un ensemble d'expressions, lors de l'application d'une règle de transformation. C'est la même idée qui a conduit à utiliser des diagrammes de décision binaires pour représenter des ensembles d'états en vérification des modèles.

Ce chapitre présente la structure de données que nous avons mise au point pour réaliser cet objectif. Nous la présentons essentiellement à un niveau abstrait, utile et suffisant pour comprendre en quoi elle permet de représenter des ensembles de termes égaux par rapport à une théorie a priori quelconque, et d'appliquer des opérations globales sur ces ensembles de termes, sans les énumérer. Toutefois, la définition abstraite de la structure de données que nous proposons ne suffit pas à démontrer

1. Expressions algébriques, formules booléennes, expressions rationnelles, pour citer quelques classes de problèmes...

son utilité. Il faut aussi qu'elle soit implémentable efficacement. Nous montrons dans ce chapitre qu'une telle implémentation existe et qu'elle est relativement sophistiquée et délicate. Pour étayer notre thèse, nous utilisons notre structure de données pour résoudre un problème de décision classique : déterminer si deux termes sont égaux par rapport à une théorie définie par un ensemble d'équations closes. Nous fournissons une étude de complexité établissant que notre méthode est aussi bonne, à ce niveau théorique, que les meilleurs algorithmes proposés dans la littérature. Nous comparons aussi expérimentalement notre méthode à certaines méthodes de la littérature. Les mesures obtenues montrent que notre méthode est plus efficace, sur les exemples choisis, que ces méthodes. Nous entreprenons de l'expliquer par le fait que les méthodes précédentes utilisent une représentation plus compliquée et moins économe en mémoire que notre structure de données. Bien qu'il soit le point de départ de tout notre travail, nous n'abordons pas, dans ce chapitre, le problème proprement dit de la simplification d'expressions car nous le traitons dans un chapitre séparé, faisant suite à celui-ci.

La suite de ce chapitre est structurée comme suit. Nous fixons d'abord la terminologie que nous utilisons pour parler des termes, des ensembles de termes et des relations définies sur des ensembles de termes. Nous précisons la sorte d'ensembles de termes qui retient notre attention. Nous passons ensuite à la définition de notre structure de données, que nous avons choisi d'appeler, faute de mieux, *collection de structures* : dans ce contexte, une "structure" est un objet "élémentaire" qui représente un ensemble de termes (grâce au concours d'autres structures d'une même collection), un "ensemble de structures" est formé de structures représentant des termes équivalents (dans une certaine théorie) et la "collection de structures" est la réunion de tous les ensembles de structures considérés dans une situation déterminée. Sur cette base, nous définissons les *opérations* relatives aux collections de structures. Chaque opération transforme une collection de structures en une autre afin d'exprimer une opération sur les ensembles de termes représentés par la collection de structures : ajouter des termes aux ensembles, décider si des termes sont dans le même ensemble, ajouter des contraintes d'égalité entre les termes de deux ensembles. Nous démontrons chaque fois que l'opération sur la collection de structures réalise effectivement l'opération voulue sur les ensembles de termes représentés par la collection de structures. Nous passons ensuite à la description de l'implémentation concrète de notre structure de données. Cette implémentation a été réalisée effectivement en utilisant le langage de programmation Java mais nous en donnons une description de plus haut niveau. Nous en déduisons les informations sur la complexité théorique des différentes opérations. Nous avons alors

tous les outils en main pour appliquer les collections de structures au problème de décision consistant à déterminer si deux termes clos sont égaux dans une théorie définie par un ensemble d'équations closes. Notre algorithme se résume à appliquer une certaine combinaison simple des opérations sur les collections de structures, à chaque équation définissant la théorie. Sa complexité théorique est facilement déterminée à partir de celle des opérations sur les collections de structures. Nous comparons expérimentalement notre algorithme, transcrit en Java, à plusieurs algorithmes publiés dans la littérature, également transcrits en Java, par nos soins, en réutilisant une grande partie de l'implémentation Java de nos collections de structures. Nous montrons, d'une part que les résultats expérimentaux sont cohérents avec les résultats de complexité "théoriques" et, d'autre part, que notre algorithme utilisant les collections de structures est plus efficace que nos transcriptions Java des algorithmes utilisés pour la comparaison. Nous expliquons cette différence par le fait que les collections de structures fournissent une représentation plus compacte des ensembles de termes que les structures utilisées par les autres algorithmes : graphes acycliques (Dags) et structures Union-Find. De plus, notre algorithme manipule directement les ensembles de structures et non les termes individuels. Nous complétons ensuite cette étude en évoquant d'autres notions liées à nos collections de structures telles que les automates d'arbres et les règles de réécriture.

Une version préliminaire du travail présenté dans ce chapitre a été publiée dans [LA16]. Une présentation des premiers résultats obtenus pour son application au problème de la simplification des expressions a fait l'objet de l'article [LA15].

2.2 Ensembles de termes et relations entre termes

2.2.1 Notions préliminaires, rappels et conventions concernant les termes

La notion de terme est suffisamment connue pour qu'il soit superflu de la définir une "nouvelle" fois, ici. Les définitions ci-dessous sont donc plutôt des rappels, pour fixer notre terminologie.

Définition 1 (Terme et sous-termes directs)**.** Un *terme* est soit un terme indécomposable, c'est-à-dire une *constante* ou une *variable*, soit un *terme composé*, formé d'un symbole de fonction et d'une liste non vide de termes plus simples. Ces termes sont appelés *sous-termes directs* du terme composé.

Définition 2 (Sous-termes d'un terme)**.** On appelle *sous-terme* d'un terme, ce terme lui-même et tous les sous-termes de ses sous-termes directs (s'il est composé).

Pour des raisons de simplicité, à la fois dans l'exposé et dans l'implémentation des collections de structures, nous ne considérons que des termes *binaires* de la forme $f(t_1, t_2)$, c'est-à-dire composé du symbole de fonction f et de la liste de sous-termes directs (t_1, t_2), avec, de plus, une seule constante, notée *null*. Cette restriction ne diminue pas la généralité de notre travail car une constante a peut être représentée par le terme binaire $a(null, null)$ et un terme *unaire* $h(a)$ par le terme binaire $h(a(null, null), null)$. De plus, les termes possédant plus de deux sous-termes directs peuvent être représentés en introduisant un symbole de fonction binaire spécial pour construire des listes de termes. Ainsi, le terme $g(a, b, c)$, pourra être représenté, par exemple, par $g(.(a(null, null), .(b(null, null), c(null, null))))$. En pratique, cependant, nous utilisons des notations abrégées telles que a, $h(a)$ ou $g(h(a), b$, $f(a, b))$ pour écrire des exemples. De même, au niveau des applications pratiques, sur machine, nous laissons à l'analyseur syntaxique le soin de traduire les termes sous forme restreinte et nous affichons les termes calculés sous forme abrégée. Ceci nous permet aussi de traiter, outre les symboles de fonction au sens strict, des opérateurs tels que $+$, $*$ ou autres.

Définition 3 (Terme clos)**.** Un terme *clos* est un terme qui ne contient pas de variable, autrement dit, dont aucun sous-terme n'est une variable.

Dans ce chapitre, nous ne considérons, sauf mention explicite du contraire, que des termes clos.

Note 1. La distinction syntaxique entre constante et variable peut être source de confusion. Par exemple, lorsqu'on simplifie une expression contenant des "variables", celles-ci sont, en général, traitées comme des constantes car les règles de simplification ne permettent normalement pas de substituer une autre expression à une "variable" de l'expression à simplifier. Au contraire, les variables utilisées pour exprimer les règles de simplification sont de "vraies variables" car elles peuvent être remplacées par des expressions complexes.

2.2.2 Relations binaires sur les termes, congruences et complétion congruente

Nous supposons donné un *ensemble $\mathcal{T}$ de tous les termes*, complètement déterminé par l'ensemble, supposé fixé, de tous les symboles de fonction (binaires). Tous les ensembles de termes T considérés ci-après sont donc des sous-ensembles de cet ensemble (unique).

Définition 4 (Ensemble de termes complet pour les sous-termes)**.** Soit T un ensemble de termes. Nous disons que T est *complet pour les sous-termes* si tout sous-terme d'un terme appartenant à T lui appartient aussi.

Remarquons que, avec nos conventions sur la forme des termes, tout ensemble de termes T, non vide et complet pour les sous-termes, contient au moins le terme *null*, puisque n'importe quel terme contient ce terme.

Définition 5 (Relation entre termes)**.** Une *relation* (binaire) ρ entre termes est une partie de $\mathcal{T} \times \mathcal{T}$. Nous abrégeons l'affirmation $\langle s, t\rangle \in \rho$ en écrivant : $s\,\rho\,t$. Nous appelons *domaine* de la relation ρ, le plus petit ensemble de termes T, tel que $\rho \subseteq T \times T$. Nous le notons $dom(\rho)$. Cet ensemble vérifie l'égalité suivante :

$$dom(\rho) \;=\; \{t \in \mathcal{T} \mid \exists s \in \mathcal{T} \;:\; s\,\rho\,t \quad \text{ou} \quad t\,\rho\,s\}.$$

Définition 6 (Relation d'équivalence entre termes)**.** Nous disons qu'une relation ρ entre termes est une *relation d'équivalence* si elle est *réflexive*, *symétrique* et *transitive* sur son domaine. Ces propriétés s'expriment comme suit :

1. (Réflexivité) $\forall t \in \mathcal{T} : t \in dom(\rho) \Longrightarrow t\,\rho\,t.$
2. (Symétrie) $\forall s, t \in \mathcal{T} :\; s\,\rho\,t \Longrightarrow t\,\rho\,s.$
3. (Transitivité) $\forall s, t, u \in \mathcal{T} :\; (s\,\rho\,u \;\text{ et }\; u\,\rho\,t) \Longrightarrow s\,\rho\,t.$

Définition 7 (Relation entre termes fonctionnellement complète)**.** Soit T, un ensemble de termes et soit ρ, une relation telle que $T = dom(\rho)$. Nous disons que la relation ρ est *fonctionnellement complète* si l'implication ci-dessous est vérifiée quels que soient les termes $s_1, s_2, t_1, t_2 \in \mathcal{T}$ et le symbole de fonction f :

$$\left.\begin{array}{c} s_i \,\rho\, t_i \;\; (i = 1, 2) \\ f(s_1, s_2) \in T \end{array}\right\} \Longrightarrow f(s_1, s_2) \;\rho\; f(t_1, t_2).$$

Définition 8 (Congruence)**.** Avec les mêmes notations que ci-dessus, nous disons que la relation ρ entre termes est une *congruence* si elle est fonctionnellement complète et si c'est une relation d'équivalence.

Notation 1. Lorsque la relation ρ est une *congruence*, nous utilisons souvent le symbole $\sim$ à la place du symbole ρ pour la désigner. Et nous disons que $\sim$ *est une congruence sur* $T = dom(\sim)$.

On vérifie facilement que l'intersection d'un ensemble non vide de congruences est une congruence. De plus, l'ensemble $\mathcal{T} \times \mathcal{T}$ est une congruence qui contient toutes les relations. Par conséquent, la définition suivante a un sens.

Définition 9 (Complétion congruente d'une relation)**.** Soit ρ, une relation entre termes. Nous appelons *complétion congruente* de la relation ρ, la plus petite congruence qui la contient. Nous notons cette relation $\sim_\rho$.

Cette définition donne évidemment lieu aux deux théorèmes suivants.

Théorème 3. *Une relation ρ entre termes est une congruence si et seulement si on a* : $\sim_\rho \;=\; \rho$.

Théorème 4. *Soient ρ et ρ' deux relations entre termes. On a* :

$$\rho \;\subseteq\; \rho' \quad \Longrightarrow \quad \sim_\rho \subseteq \sim_{\rho'} .$$

La définition 9 ne fournit pas de moyen direct pour démontrer certaines propriétés de la complétion congruente d'une relation ρ. Nous introduisons dès lors une nouvelle notion qui nous permettra de raisonner par induction pour prouver ces propriétés.

Définition 10 (Preuves d'affirmations telles que $s \sim_\rho t$ ou $t \in dom(\sim_\rho)$)**.**

1. Soit ρ une relation entre termes et soient s et t des termes. Nous appelons *preuve de l'affirmation* $s \;\sim_\rho\; t$ toute suite finie d'affirmations

   $$s_1 \;\sim_\rho\; t_1,\; s_2 \;\sim_\rho\; t_2,\; \ldots \;,\; s_i \;\sim_\rho\; t_i,\; \ldots \;,\; s_n \;\sim_\rho\; t_n \qquad (n \geq 1),$$

   telle que

(a) $s = s_n$ et $t = t_n$;

(b) chaque affirmation $s_i \sim_\rho t_i$ peut être obtenue soit comme conséquence du fait que $s_i \, \rho \, t_i$, soit en appliquant la règle du modus ponens à une des quatre implications énoncées dans les définitions 6 et 7, en utilisant comme prémisses une ou plusieurs affirmations $s_j \sim_\rho t_j$ telles que $j < i$. On considère aussi qu'une telle affirmation rend vraie chacune des prémisses $s_j \in dom(\sim_\rho)$ et $t_j \in dom(\sim_\rho)$.

2. De même, nous appelons *preuve de l'affirmation* $t \in dom(\sim_\rho)$, toute preuve d'une affirmation d'une des deux formes $s \sim_\rho t$ ou $t \sim_\rho s$.

Le théorème suivant montre que la définition 10 fournit un moyen concret de construire la relation $\sim_\rho$.

Théorème 5. *Soit ρ une relation entre termes et soient s et t des termes. L'affirmation $s \sim_\rho t$ est vraie selon la définition 9 si et seulement si il existe une preuve de $s \sim_\rho t$, au sens de la définition 10. Il en va de même pour l'affirmation $t \in dom(\sim_\rho)$.*

Preuve du théorème 5. Pour démontrer le théorème 5, nous montrons d'abord que, pour toute congruence $\sim$ contenant ρ, on a $s \sim t$, chaque fois qu'il existe une preuve de $s \sim_\rho t$. Ensuite, nous montrons que la relation $\sim'$ définie par $s \sim' t$ si et seulement si il existe une preuve de $s \sim_\rho t$, est une congruence. Ces deux résultats montrent que $\sim'$ est la plus petite congruence contenant ρ et donc que le théorème 5 est vrai.

1. Soit $\sim$, une congruence qui contient ρ. On démontre immédiatement par récurrence que toutes les affirmations $s_i \sim_\rho t_i$ d'une preuve de $s \sim_\rho t$ sont telles que $s_i \sim t_i$, par construction même d'une telle preuve. Par exemple, si l'affirmation $t_i \sim_\rho t_i$ est obtenue grâce à la règle de réflexivité de la définition 6, la preuve doit contenir une affirmation de la forme $s_j \sim_\rho t_j$ où $t_j = t_i$ ou $s_j = t_i$ et $j < i$. Par hypothèse de récurrence, cette affirmation est vraie. Donc, on a : $t_i \in dom(\sim)$, par définition du domaine d'une relation entre termes. Finalement, on a : $t_i \sim t_i$ par application du modus ponens à la règle de réflexivité.
2. Soit la relation $\sim'$ définie ci-dessus. On vérifie facilement qu'elle possède toutes les propriétés d'une congruence. Montrons, par exemple, la transitivité. Soient s, t et u des termes tels qu'on ait $s \sim' u$ et $u \sim' t$. Par définition, de la relation $\sim'$, il existe une preuve de $s \sim_\rho u$ et une preuve de $u \sim_\rho t$. On construit une preuve de

$s \sim_\rho t$ en interclassant (de manière quelconque) les deux preuves précédentes et en y ajoutant $s \sim_\rho t$ comme affirmation finale. On a donc bien $s \sim' t$. □

Nous utilisons maintenant la notion de preuve introduite dans la définition 5 pour démontrer plusieurs propriétés importantes des congruences.

Théorème 6. *Soit ρ, une relation entre termes telle que $dom(\rho) = T$. Posons $\tilde{T} = dom(\sim_\rho)$. Pour tout terme $f(t_1, t_2)$ appartenant à $\tilde{T} \setminus T$, il existe un terme $f(s_1, s_2)$ appartenant à T tel que $s_i \sim_\rho t_i$ $(i = 1, 2)$. (On a donc, évidemment, aussi : $f(s_1, s_2) \sim_\rho f(t_1, t_2)$.)*

Preuve du théorème 6. Soit un terme $f(t_1, t_2)$ appartenant à $\tilde{T} \setminus T$. Il existe une preuve de l'affirmation $f(t_1, t_2) \in dom(\sim_\rho)$ dans laquelle le terme $f(t_1, t_2)$ ne figure que dans la dernière affirmation de la preuve. Cette affirmation doit être obtenue par application de la règle de complétude fonctionnelle, car les autres règles n'introduisent que des termes appartenant à T ou figurant dans une affirmation précédente de la preuve. Donc, la preuve considérée contient des preuves plus courtes d'une affirmation de la forme $f(u_1, u_2) \in dom(\sim_\rho)$ et des deux affirmations $u_i \sim_\rho t_i$ $(i = 1, 2)$ correspondantes. Ou bien le terme $f(u_1, u_2)$ appartient à T. Le théorème est alors démontré en posant $s_i = u_i$ $(i = 1, 2)$. Ou alors, $f(u_1, u_2)$ appartient à $\tilde{T} \setminus T$. Dans ce cas, nous pouvons supposer, par hypothèse d'induction sur la longueur des preuves, qu'il existe un terme $f(s_1, s_2) \in T$ tel que $s_i \sim_\rho u_i$ $(i = 1, 2)$. On a ensuite $s_i \sim_\rho t_i$ $(i = 1, 2)$, par la transitivité de la relation $\sim_\rho$. Ce qui termine la démonstration. □

Le théorème 6 a pour conséquence directe les deux théorèmes suivants.

Théorème 7. *Soit ρ, une relation entre termes telle que $dom(\rho) = T$. Soit $\tilde{T} = dom(\sim_\rho)$. On a :*

$$\tilde{T} \;=\; \{t \in \mathcal{T} \mid \exists s \in T \;:\; s \sim_\rho t\}.$$

Preuve du théorème 7. Il est tout d'abord évident que $\{t \in \mathcal{T} \mid \exists s \in T : s \sim_\rho t\} \subseteq \tilde{T}$, par définition du domaine de $\sim_\rho$. Pour prouver l'inclusion dans l'autre sens, il suffit de montrer que pour tout terme $t \in \tilde{T}$, il existe un terme $s \in T$ tel que $s \sim_\rho t$. C'est une conséquence du théorème 6 sauf dans le cas où $t = null$. Dans ce cas, on a nécessairement $t \in T$ et donc, aussi, $t \sim_\rho t$, ce qui prouve l'affirmation, en choisissant $s = t$. □

Lorsque le domaine de la relation ρ est complet pour les sous-termes, on a le résultat suivant qui est plus fort que les théorèmes 6 et 7.

Théorème 8. *Soit T, un ensemble de termes complet pour les sous-termes et soit ρ, une relation telle que $dom(\rho) = T$. Posons $\tilde{T} = dom(\sim_\rho)$. Soit $f(t_1, t_2) \in \tilde{T}$. Alors, il existe un terme $f(s_1, s_2) \in T$ tel que*

$$f(s_1, s_2) \sim_\rho f(t_1, t_2) \text{ et } s_i \sim_\rho t_i \, (i = 1, 2).$$

Preuve du théorème 8. Soit $t \in \tilde{T}$.

1. Si $t \in \tilde{T} \setminus T$, le résultat annoncé est simplement ce qui est affirmé par la théorème 6.
2. Si, au contraire, $f(t_1, t_2) \in T$, il suffit de choisir $f(s_1, s_2)$ égal à $f(t_1, t_2)$ pour établir la proposition car $t_i \sim_\rho t_i$ $(i = 1, 2)$ puisque $t_1, t_2 \in T$ dû au fait que T est complet pour les sous-termes. □

Le théorème suivant est important pour la suite (voir preuve du lemme 6).

Théorème 9. *Soit T, un ensemble de termes complet pour les sous-termes et soit ρ, une relation telle que $dom(\rho) = T$. L'ensemble $\tilde{T} = dom(\sim_\rho)$ est aussi complet pour les sous-termes.*

Preuve du théorème 9. Soit un terme $t \in \tilde{T}$. Soit

$$s_1 \sim_\rho t_1, \; s_2 \sim_\rho t_2, \; \ldots \; , \; s_i \sim_\rho t_i, \; \ldots \; , \; s_n \sim_\rho t_n \qquad (n \geq 1),$$

une preuve de l'affirmation $t \in dom(\sim_\rho)$. Nous prouvons, par induction sur i, que tous les sous-termes des termes s_i et t_i appartiennent à $\tilde{T}$ $(1 \leq i \leq n)$.

1. Si l'affirmation $s_i \sim_\rho t_i$ est obtenue du fait que l'on a $s_i \, \rho \, t_i$, les sous-termes de s_i et t_i appartiennent à T, qui est complet pour les sous-termes, donc aussi à $\tilde{T}$.
2. Si l'affirmation $s_i \sim_\rho t_i$ est obtenue en appliquant la réflexivité, la symétrie ou la transitivité, les termes s_i et t_i figurent dans une affirmation précédente de la preuve. Leurs sous-termes appartiennent donc à $\tilde{T}$, par hypothèse d'induction.
3. Si l'affirmation $s_i \sim_\rho t_i$ est obtenue en appliquant la règle de complétude fonctionnelle, les termes s_i et t_i s'écrivent respectivement sous la forme $f(s_j, s_k)$ et $f(t_j, t_k)$ où $j, k < i$ et les affirmations $s_j \sim_\rho t_j$ et $s_k \sim_\rho t_k$ figurent dans la preuve. Par hypothèse d'induction, tous les sous-termes des termes s_j, s_k, t_j, t_k appartiennent à $\tilde{T}$. C'est donc aussi le cas pour les sous-termes de s_i et t_i. □

2.2.3 Complétion congruente versus fermeture congruente d'une relation dans $\mathcal{T}$

Une relation ρ entre termes clos peut être vue comme un ensemble *Eq*, éventuellement infini, d'équations closes (entre termes clos). Ces équations déterminent une relation d'équivalence $\cong_\rho$ sur l'ensemble $\mathcal{T}$ de tous les termes telle que $s \cong_\rho t$ si et seulement si les termes s et t sont égaux dans tous les modèles de *Eq*. Supposons que l'ensemble $T = dom(\rho)$ soit complet pour les sous-termes [2]. Nous affirmons, alors, que la complétion congruente $\sim_\rho$ de la relation ρ coïncide avec la relation $\cong_\rho$ dans tout l'ensemble $\tilde{T} = dom(\sim_\rho)$. Il s'en suit également que l'ensemble $\tilde{T}$ contient exactement tous les termes qui sont égaux à au moins un terme de T dans tous les modèles de *Eq*. Finalement, il s'avère que la relation $\cong_\rho$ est la plus petite relation sur tous les termes (de domaine égal à $\mathcal{T}$) contenant $\sim_\rho$ et fonctionnellement complète. Ce qui fournit un algorithme efficace et direct pour décider la relation $\cong_\rho$, si on sait décider la relation $\sim_\rho$. La justification de ces affirmations fait l'objet de cette sous-section.

Définition 11 (Fermeture congruente d'une relation dans l'ensemble de tous les termes)**.** Soit ρ, une relation entre termes. Nous notons $\cong_\rho$ la plus petite relation d'équivalence sur l'ensemble de *tous les termes* telle que l'on ait, quels que soient les termes s, t, $f(s_1, s_2)$, $f(t_1, t_2)$:

$$\begin{array}{rcl} s\ \rho\ t & \Longrightarrow & s\ \cong_\rho t \\ s_i\ \cong_\rho t_i \quad (i = 1, 2) & \Longrightarrow & f(s_1, s_2)\ \cong_\rho f(t_1, t_2). \end{array}$$

On voit facilement que la relation $\cong_\rho$ est une congruence sur l'ensemble de tous les termes (au sens de la définition 8). Le théorème suivant établit le lien précis existant entre les notions de complétion congruente d'une relation et celle de fermeture congruente de cette relation sur l'ensemble de tous les termes.

Théorème 10. *Soient T, un ensemble de termes, complet pour les sous-termes, et ρ, une relation telle que $T = dom(\rho)$. On a les deux égalités ci-dessous :*

$$\begin{array}{rcl} \tilde{T} & = & \{\ t \in \mathcal{T} \mid \exists s \in T : s\ \cong_\rho\ t\ \} \\ \sim_\rho & = & \{\ \langle s, t\rangle \mid s, t \in \tilde{T}\ \text{ et }\ s\ \cong_\rho t\ \}. \end{array}$$

Pour démontrer ce théorème, nous prouvons d'abord un lemme important.

2. S'il ne l'est pas, nous pouvons le compléter avec ses sous-termes, en ajoutant à la relation ρ, une paire $\langle t, t\rangle$ pour chacun de ces sous-termes. Cela ne modifie pas l'ensemble des modèles des équations.

Lemme 6. *Avec les notations habituelles, soient s et t, deux termes quelconques. Si $s \in \tilde{T}$ ou $t \in \tilde{T}$, de $s \cong_\rho t$, on peut déduire $s \sim_\rho t$.*

Preuve du lemme 6. Nous prouvons le lemme par induction sur la longueur d'une preuve de $s \cong_\rho t$. Il y a cinq cas à considérer.

1. Si $s \,\rho\, t$, on a directement $s \sim_\rho t$, par définition de $s \sim_\rho t$.
2. Si $s = t$, on a $s \in \tilde{T}$ et $t \in \tilde{T}$. D'où $s \sim_\rho t$ puisque $\sim_\rho$ est une relation d'équivalence sur $\tilde{T}$.
3. Si $s \cong_\rho t$ parce qu'il existe une preuve plus courte de $t \cong_\rho s$, le résultat est immédiat.
4. Si $s \cong_\rho t$ parce qu'il existe des preuves plus courtes de $s \cong_\rho r$ et de $r \cong_\rho t$, où r est un terme quelconque, supposons, pour fixer les idées, que $s \in \tilde{T}$. Par hypothèse d'induction, on a : $s \sim_\rho r$. D'où aussi, $r \in \tilde{T}$. Donc, par hypothèse d'induction, on peut affirmer que $r \sim_\rho t$. Finalement, par la transitivité de $\sim_\rho$, on a : $s \sim_\rho t$.
5. Si s et t s'écrivent respectivement $f(s_1, s_2)$ et $f(t_1, t_2)$ et si la preuve de $s \cong_\rho t$ est obtenue en vertu de l'implication

$$s_i \;\cong_\rho t_i \;\; (i = 1, 2) \quad \Longrightarrow \quad f(s_1, s_2) \;\cong_\rho f(t_1, t_2),$$

   supposons, pour fixer les idées, que $s \in \tilde{T}$. Alors, puisque $\tilde{T}$ est complet pour les sous-termes (théorème 9), on sait que $s_i \in \tilde{T}$ $(i = 1, 2)$. On peut donc dire, par hypothèse d'induction, que $s_i \sim_\rho t_i$ $(i = 1, 2)$. Par conséquent, puisque la relation $\sim_\rho$ est fonctionnellement complète, on a : $s = f(s_1, s_2) \sim_\rho f(t_1, t_2) = t$. □

Note 2. Nous avons utilisé, pour démontrer le lemme 6, la notion de preuve d'une affirmation de la forme $s \cong_\rho t$, sans la définir explicitement. Cette notion est évidemment similaire à celle de preuve d'une affirmation $s \sim_\rho t$ (définition 10) et nous pensons que le lecteur aura fait lui-même les adaptations nécessaires.

Remarquons que le lemme 6 implique, en particulier, que si deux termes s et t vérifient $s \cong_\rho t$, ou bien ils appartiennent tous les deux à $\tilde{T}$, ou bien aucun des deux n'y appartient. Nous pouvons maintenant donner une démonstration du théorème 10.

Preuve du théorème 10. Nous prouvons d'abord la seconde égalité :

$$\sim_\rho \;\; = \;\; \{\, \langle s, t \rangle \mid s, t \in \tilde{T} \;\text{ et }\; s \;\cong_\rho t \,\},$$

en deux étapes.

1. Prouvons que $\sim_\rho \subseteq \{ \langle s,t\rangle \mid s,t \in \tilde{T} \text{ et } s \cong_\rho t \}$.
   Comme $\sim_\rho \subseteq \tilde{T} \times \tilde{T}$, il suffit de prouver que $\sim_\rho \subseteq \cong_\rho$. Nous prouvons, par induction sur la longueur d'une preuve de $s \sim_\rho t$, que $s \sim_\rho t \implies s \cong_\rho t$, quels que soient les termes s et t. À nouveau, il y a cinq cas à envisager.
   (a) Si $s \rho t$, on a directement $s \cong_\rho t$, par définition de $\cong_\rho$.
   (b) Si $s = t$, on a directement $s \cong_\rho t$, puisque $\cong_\rho$ est une relation d'équivalence sur l'ensemble de tous les termes.
   (c) Si $s \sim_\rho t$, parce qu'il existe une preuve plus courte de $t \sim_\rho s$, le résultat est immédiat.
   (d) Si $s \sim_\rho t$, parce qu'il existe des preuves plus courtes de $s \sim_\rho r$ et de $r \sim_\rho t$, où $r \in \tilde{T}$, on a, par hypothèse d'induction : $s \cong_\rho r$ et $r \cong_\rho t$. Donc, par la transitivité de $\cong_\rho$, on a : $s \cong_\rho t$.
   (e) Si s et t s'écrivent respectivement $f(s_1,s_2)$ et $f(t_1,t_2)$ et si la preuve de $s \sim_\rho t$ est obtenue en vertu de la complétude fonctionnelle de $\sim_\rho$ (voir définition 7), il existe des preuves plus courtes des relations $s_i \sim_\rho t_i$ $(i = 1,2)$. D'où, par hypothèse d'induction : $s_i \cong_\rho t_i$ $(i = 1,2)$. Il s'en suit, vu la définition 11, que $s \cong_\rho t$.
2. Prouvons que $\{ \langle s,t\rangle \mid s,t \in \tilde{T} \text{ et } s \cong_\rho t \} \subseteq \sim_\rho$.
   Soient deux termes s et t tels que $s,t \in \tilde{T}$ et $s \cong_\rho t$. On a, par application directe du lemme 6 : $s \sim_\rho t$.

Prouvons maintenant la première partie du théorème :

$$\tilde{T} = \{ t \in \mathcal{T} \mid \exists s \in T : s \cong_\rho t \}$$

Cette égalité découle immédiatement du théorème 7 et du fait que les propositions $(\exists s \in T : s \cong_\rho t)$ et $(\exists s \in T : s \sim_\rho t)$ sont équivalentes, en vertu du lemme 6 et du fait que $\sim_\rho \subseteq \cong_\rho$. □

Nous énonçons maintenant un théorème qui dit, en quelque sorte, que la relation $\sim_\rho$ contient autant d'information que $\cong_\rho$ ou, autrement dit, qu'il existe un algorithme direct pour décider la relation $\cong_\rho$ s'il en existe un pour décider $\sim_\rho$.

Théorème 11. *Soient T, un ensemble de termes, complet pour les sous-termes, et ρ, une relation telle que $dom(\rho) = T$. La relation $\cong_\rho$ est égale à la plus petite relation $\simeq_\rho$ sur l'ensemble de tous les termes telle que l'on ait, pour tous les termes s, t, $f(s_1,s_2)$, $f(t_1,t_2)$:*

$$\begin{array}{rcl} s \sim_\rho t & \implies & s \simeq_\rho t \\ s_i \simeq_\rho t_i \quad (i = 1,2) & \implies & f(s_1,s_2) \simeq_\rho f(t_1,t_2). \end{array}$$

Note 3. Il est important de bien noter que l'énoncé du théorème 11 *ne dit pas* que la relation $\simeq_\rho$ est une relation d'équivalence, mais, seulement, que c'est une relation *quelconque*. Donc, prouver le théorème revient à prouver que les conditions qui lui sont imposées impliquent que c'est nécessairement une relation d'équivalence, puisque la relation $\cong_\rho$ en est une et qu'elle vérifie les deux implications de l'énoncé.

Pour mener à bien la démonstration du théorème 11, nous utilisons la notion suivante.

Définition 12 (Poids d'un terme associé à une relation ρ)**.** Soit T, un ensemble de termes, complet pour les sous-termes. Soit ρ une relation telle que $T = dom(\rho)$. Soit aussi $\tilde{T} = dom(\sim_\rho)$. À chaque terme $t \in \mathcal{T}$, nous associons un entier naturel $w_\rho(t)$, comme suit, par induction sur la structure des termes.

1. Si $t \in \tilde{T}$, $w_\rho(t) = 0$.
2. Si $t \notin \tilde{T}$, $w_\rho(t) = \max(w_\rho(t_1), w_\rho(t_2)) + 1$, où t s'écrit $f(t_1, t_2)$.

Cette définition est cohérente parce que $\tilde{T}$ contient nécessairement le terme spécial *null*. Nous avons ensuite besoin du lemme suivant.

Lemme 7. *Avec les notations du théorème 11 et de la définition 12, on a, quels que soient les termes s et t : $s \simeq_\rho t \implies w_\rho(s) = w_\rho(t)$.*

Preuve du lemme 7. On vérifie facilement le lemme par induction sur la longueur d'une preuve de $s \simeq_\rho t$, qui utilise uniquement les deux implications définissant $\simeq_\rho$, dans l'énoncé du théorème 11. □

Preuve du théorème 11. Nous démontrons que la relation $\simeq_\rho$ est réflexive, symétrique et transitive.

1. Soit t, un terme quelconque. Si $w_\rho(t) = 0$, on a $t \sim_\rho t$ car $t \in \tilde{T}$. On a donc, aussi $t \simeq_\rho t$, par définition de $t \simeq_\rho t$. Si $w_\rho(t) > 0$, t s'écrit sous la forme $f(t_1, t_2)$ et $w_\rho(t_1), w_\rho(t_2) < w_\rho(t)$. On peut donc supposer par hypothèse d'induction sur $w_\rho(t)$ que $t_i \simeq_\rho t_i$ $(i = 1, 2)$. D'où il vient que $t \simeq_\rho t$, par la deuxième implication de la définition de $\simeq_\rho$. Ceci prouve que la relation $\simeq_\rho$ est réflexive.
2. Le cas de la symétrie se démontre assez semblablement au cas précédent en observant, d'après le lemme 7, que $s \simeq_\rho t$ implique $w_\rho(s) = w_\rho(t)$.
3. Montrons en détail la transitivité. Soient des termes s, r, t et supposons qu'on ait $s \simeq_\rho r$ et $r \simeq_\rho t$. D'après le lemme 7, on a : $w_\rho(s) = w_\rho(r) = w_\rho(t)$. Distinguons alors deux cas.

(a) Si $w_\rho(s) = 0$, il est nécessaire, d'après la définition de $\simeq_\rho$, d'avoir $s \sim_\rho r$ et $r \sim_\rho t$. D'où $s \sim_\rho t$, par transitivité de $\sim_\rho$. D'où, finalement, $s \simeq_\rho t$, par définition de $\simeq_\rho$.

(b) Si $w_\rho(s) > 0$, les relations $s \simeq_\rho r$ et $r \simeq_\rho t$ ne peuvent avoir été obtenue qu'en vertu de la seconde implication de la définition de $\simeq_\rho$, il s'en suit que les termes s, r et t s'écrivent respectivement sous la forme $f(s_1, s_2)$, $f(r_1, r_2)$ et $f(t_1, t_2)$, pour le même symbole de fonction f, et que l'on a, en outre, $s_i \simeq_\rho r_i$ et $r_i \simeq_\rho t_i$ $(i = 1, 2)$. D'après la définition du poids d'un terme, on a aussi $w_\rho(s_i) = w_\rho(r_i) = w_\rho(t_i) < w_\rho(s) = w_\rho(r) = w_\rho(t)$ $(i = 1, 2)$. On peut donc supposer, par hypothèse d'induction sur $w_\rho(s)$, que $s_i \simeq_\rho t_i$ $(i = 1, 2)$. D'ou, il vient : $s = f(s_1, s_2) \simeq_\rho f(t_1, t_2) = t$, par la deuxième implication de la définition de $\simeq_\rho$. □

L'algorithme pour décider si $s \cong_\rho t$ s'énonce donc comme suit.

1. Si $s \in \tilde{T}$ et $t \in \tilde{T}$, répondre oui, si $s \sim_\rho t$; répondre non, sinon.
2. Si $s \notin \tilde{T}$ et $t \notin \tilde{T}$ et s et t s'écrivent respectivement $f(s_1, s_2)$ et $f(t_1, t_2)$ (pour le même symbole de fonction f), répondre oui, si $s_1 \cong_\rho t_1$ et $s_2 \cong_\rho t_2$; répondre non, sinon.
3. Dans tous les autres cas, répondre non.

L'algorithme ci-dessus est récursif et sa terminaison est évidente par induction sur la structure (ou sur le poids) des termes. Il suppose qu'il existe un algorithme pour décider $s \sim_\rho t$. Cet algorithme sera fourni par la structure de données présentée à la section 2.3.

2.2.4 Présentation d'une relation entre termes

La notion que nous allons définir maintenant permet de représenter une relation entre termes par une famille d'ensembles de termes. Elle a un rôle clé dans la suite.

Définition 13 (Présentation d'une relation entre termes)**.** Soit $(T_i)_{i \in I}$, une famille d'ensembles de termes. Nous disons que $(T_i)_{i \in I}$ est une *présentation* de la relation ρ, telle que $dom(\rho) = T$ si les deux conditions ci-dessous sont vérifiées :

1. $T = \bigcup_{i \in I} T_i$;
2. $(s \ \rho \ t \iff \exists i \in I : s, t \in T_i) \quad (\forall s, t \in \mathcal{T})$

On vérifie immédiatement qu'une relation ρ possède une présentation si et seulement si elle est réflexive et symétrique. On démontre facilement le théorème suivant :

Théorème 12. *Soit $(T_i)_{i\in I}$ une présentation d'une relation ρ. La relation ρ est une relation d'équivalence si l'ensemble $\{T_i \mid i \in I\}$ est une partition de $T = dom(\rho)$.*

Note 4. L'inverse n'est pas toujours vrai.

2.3 Collections de structures

Nous présentons maintenant une structure de données permettant de représenter de manière compacte les relations entre termes étudiées à la section précédente ; en particulier, les congruences. Cette structure de données peut donc être utilisée pour résoudre des problèmes de décision tels que l'égalité des termes dans certaines théories équationnelles. Elle est aussi bien adaptée au problème de la simplification des expressions car elle permet de représenter de grands ensembles d'expressions égales à une expression donnée, puis de choisir efficacement dans cet ensemble une expression de taille minimale selon une mesure donnée.

L'utilité de cette structure de données, que nous nommons *collection de structures*, est une conséquence du fait qu'elle peut aussi être implémentée efficacement, ce que nous démontrons dans la section 2.4. Cependant, nous commençons par en donner une définition "abstraite", de "haut niveau", qui permet de bien la comprendre et de démontrer ses principales propriétés.

2.3.1 Structures, ensembles et collections de structures

Définition 14 (Identifiants d'ensembles de structures)**.** Nous supposons donné un ensemble infini $\mathcal{I}$ dont les éléments sont appelés *identifiants d'ensembles de structures*. Un sous-ensemble de cet ensemble est généralement désigné par la lettre I.

Bien que la nature exacte de l'ensemble $\mathcal{I}$ soit sans importance d'un point de vue "théorique", il est pratique de le choisir égal à l'ensemble des entiers naturels. C'est ce que nous faisons pour écrire les exemples et pour réaliser l'implémentation pratique.

Définition 15 (Structures)**.** Nous appelons *structure* un objet [3] de la forme $f(i_1, i_2) : i$ où f est un symbole de fonction et i_1, i_2 et i sont des identifiants d'ensembles de structures.

L'objet $f(i_1, i_2)$ est appelé la *clé* de la structure, tandis que l'élément i est appelé l'*identifiant de l'ensemble auquel la structure appartient*.

Définition 16 (Collections de structures)**.** Nous appelons *collection de structures* la donnée d'un nombre fini de structures, au sens de la définition 15. Nous utilisons la lettre E pour désigner une telle collection de structures.

3. Un mathématicien écrirait peut-être : un quadruplet.

Définition 17 (Ensembles de structures)**.** Soit E, une collection de structures. Soit I, l'ensemble de tous les identifiants d'ensembles de structures figurant dans les structures de E. Soit $i \in I$. Nous appelons ensemble des structures de E identifié par i, l'ensemble de toutes les structures de E qui sont de la forme $f(i_1, i_2) : i$ (avec f, i_1 et i_2, quelconques). Nous notons cet ensemble E_i.
On a donc : $E = \bigcup_{i \in I} E_i$.

Note 5. Dans la définition 17, nous nous écartons de l'usage habituel en donnant au mot *ensemble* une signification très spécifique. Un ensemble de structures E_i représente en fait un ensemble de termes "égaux" (liés par une certaine relation) et son identifiant i sert à identifier cet ensemble de termes, comme on va le voir, plus bas.

Pour être cohérent avec les conventions choisies pour les termes, nous n'utilisons, pour former des structures, que des symboles de fonction binaires. De plus, nous supposons avoir choisi, dans l'ensemble $\mathcal{I}$, un identifiant particulier que nous notons i_{null}. Cet identifiant représente l'ensemble de termes $\{null\}$. On peut supposer que l'on a toujours $i_{null} \in I$ et que $E_{i_{null}} = \{null : i_{null}\}$, pour toute collection de structures E. (Il y a donc toujours une et une seule structure qui n'est pas de la forme $f(i_1, i_2) : i$, dans une collection de structures.) À nouveau, comme pour les termes, ces conventions sont seulement de nature à simplifier l'exposé théorique et l'implémentation pratique. Dans les exemples, nous omettrons de faire référence à l'identifiant i_{null} et nous écrirons, par exemple, $a : i$, au lieu de $a(i_{null}, i_{null}) : i$, et $h(i_1) : i$, au lieu de $h(i_1, i_{null}) : i$.
Nous pouvons maintenant écrire ce qu'une collection de structures représente exactement.

Définition 18 (Dénotation (concrète) d'une collection de structure)**.** Soit $E = \bigcup_{i \in I} E_i$, une collection de structures. Nous appelons *dénotation concrète* (ou simplement *dénotation*) de E, la famille d'ensembles de termes $(T_i)_{i \in I}$ telle qu'un terme $f(t_1, t_2)$ appartient à l'ensemble T_i si et seulement si il existe une structure $f(i_1, i_2) : i$, appartenant à E_i telle que $t_j \in T_{i_j}$ $(j = 1, 2)$ (quel que soit $i \in I$).

La définition ci-dessus est cohérente et définit de manière unique la dénotation concrète d'une collection de structures car elle fournit un algorithme pour tester si un terme appartient à un quelconque des ensembles T_i. Donc, chaque ensemble T_i est défini de manière unique.

Exemple 1. Considérons l'ensemble de structures $E = E_1 \cup E_2$ où E_1 et E_2 sont définis comme suit :

$$E_1 = \{f(1, 2) : 1,\ a : 1\} \qquad\qquad E_2 = \{b : 2\}.$$

La dénotation concrète de cet ensemble de structures est la famille d'ensembles $(T_i)_{i=1,2}$ telle que

$$T_1 = \{a,\ f(a,b),\ f(f(a,b),b),\ \ldots,\ f(\ldots f(a,b)\ldots,b),\ \ldots\} \quad T_2 = \{b\}.$$

On constate que l'ensemble de termes T_1 est infini.

On peut maintenant donner une définition plus abstraite de la dénotation d'une collection de structures qui élimine toute référence aux identifiants d'ensembles de structures.

Définition 19 (Dénotation abstraite d'une collection de structure)**.** Soit E, une collection de structures et soit $(T_i)_{i\in I}$, sa dénotation concrète. Nous appelons *dénotation abstraite* de E la relation ρ entre termes, dont $(T_i)_{i\in I}$ est une présentation (voir définition 13).

La démonstration du théorème suivant est une conséquence directe des deux définitions précédentes.

Théorème 13. *Soit E, une collection de structures et soit ρ, sa dénotation abstraite. L'ensemble $T = dom(\rho)$ est complet pour les sous-termes.*

Il est possible que certaines structures d'une collection de structures ne représentent aucun terme (au sein de cette collection de structures). Pour se prémunir de cette regrettable possibilité, nous donnons la définition suivante.

Définition 20 (Collection de structures bien formée)**.** Soit E, une collection de structures. Nous disons que cette collection de structures est bien formée si l'on peut énumérer l'ensemble I de ses identifiants sous la forme d'une suite $i_1, i_2, \ldots, i_n$, de telle sorte que $i_1 = i_{null}$ et que pour tout l tel que $1 < l \leq n$, l'ensemble de structures E_{i_l} contienne une structure $f(i_j, i_k) : i_l$ où $j, k < l$.

Théorème 14. *Soit E, une collection de structures et soit $(T_i)_{i\in I}$, sa dénotation concrète. La collection E est bien formée si et seulement si aucun des ensembles T_i n'est vide ($i \in I$).*

Preuve du théorème 14.

1. La condition est nécessaire.
   Supposons qu'il existe une suite $i_1, i_2, \ldots, i_n$, telle que définie dans l'énoncé. On démontre par induction sur l que chaque ensemble T_{i_l} est non vide. C'est clairement vrai pour $l = 1$ car $T_{i_1} = \{null\}$. C'est vrai aussi pour $l > 1$ car on peut supposer, par hypothèse d'induction, que T_{i_j} contient un terme t_j et T_{i_k}, un terme t_k : donc, puisque E_{i_l} contient la structure $f(i_j, i_k) : i_l$, T_{i_l} contient le terme $t_l = f(t_j, t_k)$.

2. La condition est suffisante.

   Nous associons d'abord à chaque terme t un nombre entier naturel $h(t)$ défini par $h(null) = 0$ et $h(f(t_1, t_2)) = \max(h(t_1), h(t_2)) + 1$. Appelons $h(t)$ la hauteur de t. Comme chacun des ensembles T_i est non vide, nous pouvons associer à chaque identifiant i, un nombre h_i, égal au minimum des hauteurs des termes de T_i. Nous pouvons ensuite ranger tous les identifiants de I en une suite $i_1, i_2, \ldots, i_n$ telle que $1 \leq j < k \leq n$ implique $h_{i_j} \leq h_{i_k}$. Cette suite vérifie les conditions de la définition 20 car, nécessairement, $h_{i_1} = 0$ et, si $l > 0$, l'ensemble T_{i_l} contient un terme de taille h_{i_l} et de la forme $f(t_j, t_k)$ où il existe une structure $f(i_j, i_k) : i_l \in E_{i_l}$ telle que $t_j \in T_{i_j}$ et $t_k \in T_{i_k}$. Clairement, $h_{i_j} \leq h(t_j) < h_{i_l}$ et $h_{i_k} \leq h(t_k) < h_{i_l}$. Donc, on a obligatoirement, $j, k < l$. □

Toutes les collections de structures considérées dans la suite sont bien formées. D'après la définition 19, une collection de structures représente, en général, une relation ρ entre termes, dont le domaine $T = dom(\rho)$ est complet pour les sous-termes et qui est réflexive et symétrique. Nous nous intéressons particulièrement aux relations ρ qui sont des congruences. D'où la définition et les deux théorèmes qui suivent.

Définition 21 (Collection de structures normalisée)**.** Nous disons qu'une collection de structures E est *normalisée* si elle ne contient pas deux structures de la forme $f(i_1, i_2) : i$ et $f(i_1, i_2) : i'$ (possédant la même clé mais appartenant à deux ensembles de structures distincts ($i \neq i'$)).

Théorème 15. *Soit E, une collection de structures et soit $(T_i)_{i \in I}$, sa dénotation concrète. La collection E est normalisée si et seulement si l'ensemble $\{T_i \mid i \in I\}$ est une partition de $T = \bigcup_{i \in I} T_i$ et $T_i \neq T_j$ $(\forall i, j \in I : i \neq j)$.*

Preuve du théorème 15.

1. La condition est nécessaire.

   Supposons que la collection E soit normalisée. Nous prouvons, par induction sur la structure des termes, qu'un terme $t \in T$ appartient à un ensemble T_i, exactement. Soit $t = f(t_1, t_2)$, un tel terme. Par définition de la dénotation concrète de E, il existe une structure $f(i_1, i_2) : i$ de E telle que $t_j \in T_{i_j}$ $(j = 1, 2)$. Par hypothèse d'induction, chacun des termes t_j n'appartient à aucun autre ensemble que T_{i_j}. Il est donc impossible que t appartienne à un autre ensemble $T_{i'}$ $(i \neq i')$ puisque cela impliquerait que la structure $f(i_1, i_2) : i'$ appartiendrait à $E_{i'}$.

2. La condition est suffisante.

   Supposons que l'ensemble $\{T_i \mid i \in I\}$ soit une partition de $T = \bigcup_{i \in I} T_i$ et que $T_i \neq T_j$ $(\forall i, j \in I : i \neq j)$. Supposons que, contrairement à la thèse, la collection E ne soit pas normalisée. Donc, elle contient deux structures $f(i_1, i_2) : i$ et $f(i_1, i_2) : i'$ avec $(i \neq i')$. Il s'en suit que les ensembles T_i et $T_{i'}$ ne sont pas disjoint. Comme $\{T_i \mid i \in I\}$ est une partition de T, ce n'est possible que si $T_i = T_{i'}$. Mais cette possibilité est expressément exclue par l'hypothèse. La collection E doit donc être normalisée. □

Théorème 16. *Soit E, une collection de structures. Si la collection E est normalisée, sa dénotation abstraite est une congruence.*

Note 6. L'inverse n'est pas nécessairement vrai.

Preuve du théorème 16. Soit ρ, la dénotation abstraite de E et soit $T = dom(\rho)$. D'après le théorème 15, la relation ρ est une relation d'équivalence sur T. Pour montrer que c'est une congruence, il suffit de prouver qu'elle est fonctionnellement complète (voir définition 7). Soit $(T_i)_{i \in I}$, la dénotation concrète de E.

Soient des termes $s_1, s_2, t_1, t_2 \in \mathcal{T}$. Supposons que $s_j \; \rho \; t_j \, (j = 1, 2)$ et que $f(s_1, s_2) \in T$. Comme $s_j \; \rho \; t_j \; (j = 1, 2)$, il existe des identifiants i_j tels que $s_j, t_j \in T_{i_j} \; (j = 1, 2)$. Comme $f(s_1, s_2) \in T$, la collection E contient une structure de la forme $f(i'_1, i'_2) : i$ et $s_j \in T_{i'_j} \; (j = 1, 2)$. Puisque E est normalisée et que $s_j \in T_{i_j}$ et $s_j \in T_{i'_j}$, $(j = 1, 2)$, on a $: i_j = i'_j$ $(j = 1, 2)$, d'après le théorème 15. Il s'en suit que $f(s_1, s_2), f(t_1, t_2) \in T_i$ et donc que $f(s_1, s_2) \; \rho \; f(t_1, t_2)$. Ce qui montre que la relation ρ est fonctionnellement complète. □

2.3.2 Opérations sur les collections de structures

Nous avons montré, dans la sous-section précédente, qu'une collection de structures représente un ensemble de termes T, complet pour les sous-termes, muni d'une relation ρ. Intuitivement, chaque couple de la relation ρ peut être interprété comme une contrainte d'égalité entre deux termes de T. Dans le meilleur des cas, la relation ρ est une congruence sur T. (Nous la notons alors, de préférence, $\sim$.)

Nous allons maintenant introduire des opérations sur les collections de structures permettant de les créer et d'y inclure de nouveaux termes, de les modifier en y ajoutant des contraintes d'égalités, et de les normaliser pour qu'elles expriment le plus explicitement possible toutes les conséquences logiques des contraintes qui leur ont été imposées.

Pour chaque opération, nous prévoyons quatre points.

1. Une description intuitive, facile à comprendre, mais parfois incorrecte à strictement parler, de ce que réalise l'opération.
2. Une description opérationnelle de l'opération, permettant de la calculer, par exemple, à la main, sur des exemples. (La description de l'implémentation exacte des opérations fait l'objet de la section 2.4.)
3. Si la description intuitive du point 1. n'est pas suffisamment précise ou correcte, nous donnons une caractérisation exacte de l'effet de l'opération, en termes de dénotation des collections de structures. (Nous appelons cette caractérisation *la spécification* de l'opération.)
4. Nous démontrons que la description opérationnelle de l'opération est correcte par rapport à sa spécification.

2.3.2.1 L'opération *toSet*

Intuitivement, l'opération *toSet* ajoute un terme t à une collection de structures normalisée E, s'il n'y appartient pas déjà. Elle renvoie, de plus, l'identifiant i de l'ensemble de structures E_i auquel le terme appartient (après exécution de l'opération). La collection de structures reste normalisée après exécution de l'opération.

Note 7. La description ci-dessus est incorrecte puisqu'une collection de structures n'est pas un ensemble de termes. Il faut comprendre que la collection de structures E est modifiée de telle sorte que le terme t soit ajouté à l'ensemble T des termes représentés par E. Ou, plus précisément, au domaine $dom(\sim)$, de la dénotation abstraite $\sim$ de E. (Voir la notation 1.)

L'opération *toSet* se calcule comme suit.

Définition 22 (Définition opérationnelle de l'opération *toSet*)**.** L'opération *toSet* prend comme argument un terme $f(t_1, t_2)$ et une collection de structures normalisée E. Elle modifie (éventuellement) la collection de structures et renvoie un identifiant d'ensemble de structures i, comme suit.

On applique d'abord récursivement l'opération *toSet* aux sous-termes t_1 et t_2, ce qui modifie éventuellement la collection de structures E et renvoie deux identifiants i_1 et i_2. Si la collection E ne contient pas de structures de la forme $f(i_1, i_2) : i$, on en crée une, après avoir choisi une nouvel identifiant i (non utilisé par E), et on l'ajoute à E. Dans

tous les cas, E contient maintenant une structure s'écrivant $f(i_1, i_2) : i$. L'identifiant i est finalement renvoyé par l'opération.

L'effet de l'opération *toSet* est caractérisé par le théorème suivant.

Notation 2. Soit $\sim$, une congruence (voir la définition 8 et la notation 1). Nous appelons la relation $\cong_\sim$: *extension de* $\sim$ *à l'ensemble de tous les termes* et nous la notons plus simplement $\cong$.

Théorème 17 (Spécification de l'opération *toSet*)**.** *Avec les notations de la définition 22, soit* $\sim$ *la dénotation abstraite de la collection de structures* E *avant d'appliquer l'opération toSet. Soit* $T = dom(\sim)$ *et soit* S*, l'ensemble des sous-termes de* $f(t_1, t_2)$*.*

1. *La dénotation finale* $\sim'$ *de* E *vérifie les égalités suivantes :*

$$\begin{array}{rcl} T' & = & \{\ t \in \mathcal{T} \mid \exists s \in T \cup S : s \ \cong \ t\ \} \\ \sim' & = & \{\ \langle s, t\rangle \mid s, t \in T' \ \text{ et } \ s \ \cong t\ \} \end{array}$$

2. *Soit* $(T'_i)_{i\in I'}$*, la dénotation concrète finale de* E*. L'identifiant* i*, renvoyé par l'opération est tel que* $f(t_1, t_2) \in T'_i$*.*

Le théorème ci-dessus peut être réexprimé de manière plus facile à comprendre en disant que l'opération *toSet* ajoute au domaine de la dénotation abstraite de E tous les termes "égaux" (i.e. équivalents selon la relation $\cong$) aux sous-termes de $f(t_1, t_2)$ (sauf s'ils lui appartenaient déjà). Il n'est pas possible d'en ajouter moins.

Preuve du théorème 17. L'exécution de l'opération décrite dans la définition 22 ne modifie la collection de structures E qu'en lui ajoutant de nouvelles structures $f(i_1, i_2) : i$ telles que la clé $f(i_1, i_2)$ n'apparaît pas dans E et l'identifiant i n'est pas utilisé dans E. Il ne peut donc jamais y avoir deux structures avec la même clé dans E qui reste toujours une collection de structures normalisée. Par ailleurs, le fait d'ajouter une telle structure à E ne modifie pas la relation $\cong$ qui étend à tous les termes la dénotation abstraite de E. C'est une conséquence du théorème 11. Chaque ajout d'une telle structure ajoute au domaine de la dénotation de E un ensemble de termes T_i contenant exactement tous les termes $f(s_1, s_2)$ tels que $s_i \cong t_i$ $(i = 1, 2)$ où $f(t_1, t_2)$ est le terme de départ ou un de ses sous-termes, c'est-à-dire tous les termes $f(s_1, s_2)$ tels que $f(s_1, s_2) \cong f(t_1, t_2)$. Au total, on a bien, à la fin :

$$T' \ = \ \{\ t \in \mathcal{T} \mid \exists s \in T \cup S : s \ \cong \ t\ \}.$$

Le deuxième point du théorème est évident. □

Exemple 2. Soit la collection de structures $E = E_1 \cup E_2$, où :

$$E_1 = \{a : 1\} \qquad E_2 = \{b : 2,\ f(1) : 2\}.$$

Elle dénote les deux ensembles de termes, ci-dessous :

$$T_1 = \{a\} \qquad T_2 = \{b,\ f(a)\}.$$

Appliquons l'opération *toSet* au terme $f(b)$ et à cette collection de structures. Nous obtenons la collection de structures suivante :

$$E'_1 = \{a : 1\} \qquad E'_2 = \{b : 2,\ f(1) : 2\} \qquad E'_3 = \{f(2) : 3\}.$$

Elle dénote les ensembles de termes que voici :

$$T'_1 = \{a\} \qquad T'_2 = \{b,\ f(a)\} \qquad T'_3 = \{f(b),\ f(f(a))\}.$$

Nous constatons que le terme $f(b)$ a été ajouté à T, mais aussi le terme $f(f(a))$ parce que $f(b) \cong f(f(a))$.
On peut noter que l'opération renvoie l'identifiant 3.

Remarque 1. Il est possible de généraliser l'opération *toSet* pour qu'elle s'applique à un terme t comportant des variables (terme non clos). Elle reçoit alors un argument supplémentaire (une fonction) qui associe à chaque variable x de t un identifiant d'ensemble de structures i. On ajoute alors (notamment) à la (dénotation de la) collection de structures, tous les termes obtenus en remplaçant chaque variable x de t par un terme de l'ensemble de termes T_i, correspondant. Cette généralisation de l'opération *toSet* est utile pour développer des algorithmes de simplification d'expressions.

2.3.2.2 L'opération *substitute*

La deuxième opération, nommée *substitute*, sert intuitivement à ajouter une contrainte d'égalité entre les termes représentés par deux ensembles de structures E_k et E_l. (En supposant que l'on "voit" chaque couple de la dénotation abstraite ρ de E comme une contrainte d'égalité entre termes.) Cette opération ne suppose pas que E est normalisée et ne produit généralement pas une nouvelle collection normalisée, même si la collection de départ l'est.

Pour faciliter la définition de cette opération, nous introduisons une petite notation supplémentaire.

Notation 3 (Remplacement d'un identifiant dans une structure). Convenons d'utiliser le mot *struct* pour désigner une structure sans la décrire complètement. Soient k et l, deux identifiants d'ensembles de structures. Nous notons $struct[k \mapsto l]$ la structure obtenue en remplaçant chaque occurrence de l dans *struct*, par k.

Exemple 3. La structure $f(1,3) : 3[2 \mapsto 3]$ est égale à $f(1,2) : 2$.
Par ailleurs, la structure $f(2,3) : 3[2 \mapsto 3]$ est égale à $f(2,2) : 2$.

Nous définissons maintenant l'opération *substitute*.

Définition 23 (Définition opérationnelle de l'opération *substitute*)**.** L'opération *substitute* prend comme argument une collection de structures E et deux identifiants k et l, utilisés par cette collection de structures. Elle retire de E toutes les structures *struct* contenant au moins une occurrence de l et elle ajoute à E toutes les structures $struct[k \mapsto l]$ correspondantes qui ne lui appartiennent pas déjà.

Exemple 4. Appelons E la collection de structures obtenue à la fin de l'exemple 2. Pour rappel, elle se compose des trois ensembles de structures ci-dessous :

$$E_1 = \{a : 1\} \qquad E_2 = \{b : 2,\ f(1) : 2\} \qquad E_3 = \{f(2) : 3\},$$

et elle dénote les ensembles de termes suivants :

$$T_1 = \{a\} \qquad T_2 = \{b,\ f(a)\} \qquad T_3 = \{f(b),\ f(f(a))\}.$$

Appliquons l'opération *substitute* à cette collection de structures et aux identifiants 1 et 2. Toutes les occurrences de l'identifiant 2 sont remplacées par l'identifiant 1. Nous obtenons la collection de structures $E' = E'_1 \cup E'_3$, telle que :

$$E'_1 = \{a : 1,\ b : 1,\ f(1) : 1\} \qquad E'_3 = \{f(1) : 3\}.$$

Les ensembles de termes correspondants sont donc :

$$T'_1 = \{a,\ b,\ f(a),\ f(b),\ f(f(a)),\ f(f(b)),\ f(f(f(a))),\ f(f(f(b))),\ \dots\}$$

$$T'_3 = \{f(a),\ f(b),\ f(f(a)),\ f(f(b)),\ f(f(f(a))),\ f(f(f(b))),\ \dots\}.$$

Nous observons que la collection de structures n'est pas normalisée puisque les structures $f(1) : 1$ et $f(1) : 3$ ont la même clé $f(1)$. Du coup, les ensembles de termes T'_1 et T'_3 ne sont pas disjoints.

Nous pouvons maintenant caractériser sémantiquement de manière précise l'opération *substitute*.

Théorème 18 (Spécification de l'opération *substitute*)**.** *Reprenons les notations de la définition 23. Soient ρ et ρ', respectivement, les dénotations abstraites de la collection de structures E, avant et après application de l'opération. Soit encore $(T_i)_{i \in I}$, la dénotation concrète de E (avant).*

1. *La relation $\sim_{\rho'}$ est la plus petite de toutes les congruences $\sim$ qui vérifient les deux implications suivantes (pour tous les termes s, t) :*

$$\begin{aligned} s\ \rho\ t &\implies s\ \sim\ t \\ s \in T_k \text{ et } t \in T_l &\implies s\ \sim\ t \end{aligned}$$

2. *Soient $T = dom(\rho)$ et $\tilde{T} = dom(\sim_{\rho'})$. On a :*

$$\tilde{T} \ = \ \{\ t \in \mathcal{T} \mid \exists s \in T : s\ \sim_{\rho'}\ t\ \}$$

La première partie du théorème 18 signifie intuitivement que la relation ρ' contient toutes les contraintes d'égalité imposées sur les termes par la relation ρ augmentées des contraintes imposant l'égalité entre les termes de l'ensemble T_k et ceux de l'ensemble T_l. La seconde partie du théorème dit que tous les termes de $\tilde{T}$ sont équivalents à au moins un terme T, pour la relation $\sim_{\rho'}$.
Pour démontrer le théorème 18, nous démontrons d'abord plusieurs lemmes.

Lemme 8. *Soient $(T_i)_{i\in I}$ et $(T'_i)_{i\in I'}$, respectivement, les dénotations concrètes de E avant et après application de l'opération substitute à E, k et l (voir la définition 23). (On a : $I' = I\backslash\{l\}$.) On définit une fonction $\hat{}$ de I dans I', comme suit :*

$$\hat{i} \ = \ \text{si } i = l \text{ alors } k \text{ sinon } i \quad (\forall i \in I).$$

Avec cette définition, on a la propriété suivante :

$$\forall\, i \in I\ :\ T_i\ \subseteq\ T'_{\hat{i}}\ .$$

Preuve du lemme 8. Nous démontrons le lemme par induction sur la structure des termes. Fixons $i \in I$ et soit t, un terme appartenant à T_i. Montrons qu'il appartient à $T'_{\hat{i}}$. Par définition de T_i, il existe une structure $f(i_1, i_2) : i$, appartenant à E_i (avant exécution de l'opération *substitute*) et des termes t_j appartenant à T_{i_j} $(j = 1, 2)$ tels que $t = f(t_1, t_2)$. Par hypothèse d'induction, nous pouvons affirmer que t_j appartient à $T'_{\hat{i_j}}$ $(j = 1, 2)$. De plus, après exécution de l'opération *substitute*, la collection de structures E contient une structure égale à $f(i_1, i_2) : i[k \mapsto l]$, qui peut s'écrire aussi $f(\hat{i_1}, \hat{i_2}) : \hat{i}$. Donc, $f(t_1, t_2)$ appartient à $T'_{\hat{i}}$, par définition de $T'_{\hat{i}}$. □

Note 8. Dans la démonstration précédente, nous avons omis de considérer le "cas de base" où $t = null$. Dans ce cas, la démonstration est "évidente". Dans tous les théorèmes qui sont démontrés par induction sur la structure des termes, nous omettons de traiter ce cas de base et laissons au lecteur le soin de le faire s'il a "un doute".

Corollaire 1 (du lemme 8)**.** *Avec les notations du théorème 18, on a les deux implications suivantes (pour tous les termes s, t) :*

$$\begin{array}{rcl} s\ \rho\ t & \Longrightarrow & s\ \rho'\ t \\ s \in T_k \text{ et } t \in T_l & \Longrightarrow & s\ \rho'\ t \end{array}$$

La première implication peut aussi s'écrire : $\rho \subseteq \rho'$.

Preuve du corollaire 1. Le corollaire est une conséquence immédiate du lemme 8 et du fait que les familles d'ensembles de termes $(T_i)_{i\in I}$ et $(T'_i)_{i\in I'}$ sont des présentations des relations ρ et ρ', respectivement (voir les définitions 13 et 19). □

Le lemme suivant est particulièrement important pour prouver le théorème 18.

Lemme 9. *Nous reprenons les notations du théorème 18. Soit* $\sim$*, n'importe quelle congruence vérifiant les hypothèses de la première partie du théorème 18. On a le résultat suivant :*

$$\forall\, i \in I' \quad \forall t \in T'_i \quad \exists s \in T_i \ : \ s \sim t$$

Preuve du lemme 9. À nouveau, nous démontrons le lemme par induction sur la structure des termes. Pour la clarté, nous découpons la démonstration en plusieurs parties.

1. Fixons $i \in I'$ et choisissons $t \in T'_i$. Il nous faut démontrer qu'il existe un terme s tel que $s \in T_i$ et $s \sim t$.
2. Supposons que le terme t s'écrit sous la forme $f(t_1, t_2)$. Il existe donc dans E, après exécution de l'opération *substitute*, une structure s'écrivant $f(i_1, i_2) : i$ telle que $t_j \in T'_{i_j}$ $(j = 1, 2)$.
3. Il existe également dans E, avant exécution de l'opération *substitute*, une structure s'écrivant $f(h_1, h_2) : h$ où $i_j = \hat{h_j}$ $(j = 1, 2)$ et $i = \hat{h}$. (Cette structure est telle que $f(h_1, h_2) : h[k \mapsto l] = f(i_1, i_2) : i$. Elle n'est pas forcément unique.)
4. Par hypothèse d'induction, il existe des termes $s_j \in T_{i_j}$ tels que $s_j \sim t_j$ $(j = 1, 2)$.
5. Soit j, égal à 1 ou à 2. Il existe un terme $u_j \in T_{h_j}$ tel que $u_j \sim s_j$. En effet, ou bien $h_j = i_j$, dans ce cas, on peut choisir $u_j = s_j$; ou bien $h_j = l$ et $i_j = k$. Alors, on peut choisir pour u_j n'importe quel terme de T_l, puisque, par définition de $\sim$, on a que $u_j \in T_l$ et $s_j \in T_k$ implique $u_j \sim s_j$.

6. De $u_j \sim s_j$ et $s_j \sim t_j$ et, on tire d'abord $u_j \sim t_j$ ($j = 1, 2$). De plus, puisque $u_j \in T_{h_j}$ ($j = 1, 2$) et $f(h_1, h_2) : h \in E$ (avant), le terme $u = f(u_1, u_2)$ appartient à $T_h \subseteq dom(\sim)$. Donc, comme $\sim$ est une congruence, on a : $u = f(u_1, u_2) \sim f(t_1, t_2) = t$.
7. Pour conclure, nous distinguons deux cas. Comme $i = \hat{h}$, soit $h = i$, soit $h = l$ et $i = k$. Dans le premier cas, $u \in T_i$, ce qui démontre le théorème, puisque $u \sim t$. Dans le second cas, choisissons n'importe quel terme s dans T_k. On a $s \sim u$, puisque $s \in T_k$ et $u \in T_l$ implique $s \sim u$, par définition de $\sim$. Donc, par transitivité, $s \sim t$. Comme $s \in T_k$ et $k = i$, le théorème est démontré (on a : $s \in T_i$ et $s \sim t$). □

Note 9. La démonstration ci-dessus, utilise plusieurs fois le fait que les ensembles T_i sont non vides. Ce qui est vrai, puisque nous supposons la collection E bien formée au départ et que les différentes opérations maintiennent cette propriété.

Corollaire 2 (du lemme 9)**.** *Toujours avec les mêmes notations, on a :*

$$\forall\, i \in I'\ \forall s, t \in\ T'_i\ \ : s \sim t$$

Cette affirmation peut aussi s'écrire :

$$\rho'\ \subseteq\ \sim$$

Preuve du corollaire 2. Fixons $i \in I'$ et choisissons deux termes s et t dans T'_i. D'après le lemme 9, il existe des termes s' et t', dans T_i, tels que $s \sim s'$ et $t' \sim t$. Comme s' et t' appartiennent à T_i, on a : $s' \rho\, t'$, puisque $(T_i)_{i \in I}$ est une présentation de ρ. On a, dès lors, $s' \sim t'$, par définition de $\sim$. Finalement, on a, par transitivité de $\sim$: $s \sim t$.

La seconde formulation du corollaire utilise le fait que $(T'_i)_{i \in I'}$ est une présentation de ρ'. □

Il nous est maintenant facile de démontrer le théorème 18.

Preuve du théorème 18. Nous utilisons les notations et les hypothèses du théorème 18. Soit $\sim$ la plus petite congruence qui vérifie les deux implications du point 1. de l'énoncé du théorème. Nous prouvons d'abord que $\sim\ =\ \sim_{\rho'}$.

1. On a : $\sim\ \subseteq\ \sim_{\rho'}$.
   En effet, le corollaire 1 implique que la relation $\sim_{\rho'}$ est une congruence qui vérifie toutes les conditions imposées à la relation $\sim$ sauf, peut-être, la minimalité.

2. On a aussi : $\sim_{\rho'} \subseteq \sim$.
   Cette inclusion résulte immédiatement du corollaire 2, et des théorèmes 3 et 4.

Le deuxième point de l'énoncé du théorème résulte facilement du lemme 9 et du fait que $\tilde{T} = \{ t \in \mathcal{T} \mid \exists s \in T' : s \sim_{\rho'} t \}$ où $T' = dom(\rho')$. □

2.3.2.3 L'opération *normalize*

L'opération *normalize* sert, comme on le devine, à "normaliser" une collection de structures E de manière à expliciter toutes les "conséquences logiques" des contraintes d'égalités contenues dans la relation ρ dénotée abstraitement par E. En d'autres mots, elle transforme la collection de structures E de telle sorte qu'elle représente la complétion congruente $\sim_\rho$ de la relation ρ (voir la définition 9).

Définition 24 (Définition opérationnelle de l'opération *normalize*)**.** Soit E, une collection de structures (bien formée). L'opération *normalize*, appliquée à E, modifie E, comme suit.

1. Si E est déjà normalisée, on ne fait rien. (E reste inchangée.)
2. Sinon, on choisit, dans E, deux structures $f(i_1, i_2) : k$ et $f(i_1, i_2) : l$, de même clé, mais telles que $k \neq l$. On applique, ensuite, l'opération *substitute* à E et aux identifiants k et l. Finalement, on réapplique l'opération *normalize* à E.

Note 10. Il est évident que l'algorithme récursif ci-dessus peut être remplacé par une boucle simple (récursivité terminale).

Exemple 5. Appelons E, la collection de structures obtenue à la fin de l'exemple 4. On a : $E = E_1 \cup E_3$, avec :

$$E_1 = \{a : 1,\ b : 1,\ f(1) : 1\} \qquad E_3 = \{f(1) : 3\}.$$

Les ensembles de termes correspondants sont, comme on l'a vu :

$$T_1 = \{a,\ b,\ f(a),\ f(b),\ f(f(a)),\ f(f(b)),\ f(f(f(a))),\ f(f(f(b))),\ \dots\}$$

$$T_3 = \{f(a),\ f(b),\ f(f(a)),\ f(f(b)),\ f(f(f(a))),\ f(f(f(b))),\ \dots\}.$$

L'application de l'opération *normalize* à cette collection de structures choisit d'abord les structures $f(1) : 1$ et $f(1) : 3$, de même clé. Ensuite, elle applique l'opération *substitute* à E et aux identifiants 1 et 3, ce qui donne E', ci-dessous :

$$E' = E'_1 = \{a : 1,\ b : 1,\ f(1) : 1\}.$$

Dans l'exemple ci-dessus, l'opération *normalize* a seulement simplifié la collection de structures, sans déduire de nouvelle égalité entre termes. Voici un autre exemple où de nouveaux termes sont introduits et de nouvelles égalités déduites.

Exemple 6. Soit $E = E_1 \cup E_2$, la collection de structures définie par :

$$E_1 = \{a : 1,\ f(1) : 1\} \qquad\qquad E_2 = \{a : 2,\ g(2) : 2\}.$$

Les ensembles de termes correspondants sont :

$$T_1 \ = \ \{a,\ f(a),\ f(f(a)),\ f(f(f(a))),\ f(f(f(f(a)))),\ \dots\ \}$$

$$T_2 \ = \ \{a,\ g(a),\ g(g(a)),\ g(g(g(a))),\ g(g(g(g(a)))),\ \dots\ \}.$$

Cette collection contient les deux structures $a : 1$ et $a : 2$, de même clé. L'opération *substitute* est donc appliquée à E et aux identifiants 1 et 2, ce qui donne la collection :

$$E' = E'_1 = \{a : 1,\ f(1) : 1,\ g(1) : 1\}.$$

Cette collection dénote l'ensemble de termes $T' = T_1$ où

$$T_1 \ = \ \{a,\ f(a),\ g(a),\ f(f(a)),\ f(g(a)),\ g(f(a)),\ g(g(a)),\ \dots\ \}.$$

De nouveaux termes contenant à la fois le symbole de fonction f et le symbole g ont été ajoutés à l'ensemble de termes dénoté par E. Tous ces termes sont désormais égaux à tous les termes dénotés par la collection au départ.

Nous démontrons maintenant que la définition opérationnelle de l'opération *normalize* est correcte.

Théorème 19 (Spécification de l'opération *normalize*)**.** *Soit E, une collection de structures (bien formée). Soit ρ, la dénotation abstraite de E. L'application de l'opération normalize à cette collection de structures se termine et fait de E une collection de structures normalisée. De plus, la dénotation abstraite finale de E est égale à la relation $\sim_\rho$.*

Lemme 10. *Considérons une application de l'opération substitute à une collection de structures E et à deux identifiants k et l utilisés par E ($k \neq l$). Supposons que les ensembles E_k et E_l contiennent, avant application de l'opération, deux structures possédant la même clé. Soient ρ et ρ' les dénotations abstraites de E, respectivement, avant et après application de l'opération. Sous ces hypothèses, on a l'égalité suivante :*

$$\sim_\rho \ = \ \sim_{\rho'}$$

Preuve du lemme 10. D'après le corollaire 1, on a : $\rho \subseteq \rho'$. D'où, par application du théorème 4 : $\sim_\rho \subseteq \sim_{\rho'}$.
Il reste donc à montrer que $\sim_{\rho'} \subseteq \sim_\rho$. C'est vrai car la relation $\sim_\rho$ vérifie les conditions de la définition de $\sim_{\rho'}$, sauf, peut-être, la minimalité. Pour le voir, choisissons s et t, des termes quelconques.

1. Il est clair que
$$s \ \rho \ t \implies s \sim_\rho t$$
par définition de $s \sim_\rho t$.
2. Mais on a aussi
$$s \in T_k \text{ et } t \in T_l \implies s \sim_\rho t$$
car T_k et T_l contiennent au moins un terme commun, puisque les ensembles de structures E_k et E_l contiennent deux structures de même clé (avant d'appliquer l'opération *substitute*). Soit u, un tel terme. On a donc $s \ \rho \ u$ et $u \ \rho \ t$. D'où, aussi : $s \sim_\rho u$ et $u \sim_\rho t$. Donc, par transitivité : $s \sim_\rho t$. □

Nous pouvons maintenant prouver que l'opération *normalize* respecte sa spécification.

Preuve du théorème 19 (Correction de normalize).
Nous remarquons tout d'abord que l'algorithme donné dans la définition 24 se termine car le nombre d'identifiants utilisés par la collection E diminue d'une unité à chaque appel récursif. Nous pouvons dès lors prouver la correction de l'opération par induction sur le nombre d'identifiants utilisés par E.

1. Si la collection E est normalisée, sa dénotation abstraite est une congruence $\sim$ (théorème 16). Dans ce cas, l'opération *normalize* laisse E inchangé, ce qui est correct puisque $\sim = \sim_\sim$ (théorème 3).
2. Si la collection E n'est pas normalisée, l'opération choisit, dans E, deux structures $f(i_1, i_2) : k$ et $f(i_1, i_2) : l$, de même clé, et telles que $k \neq l$. Elle applique, alors, l'opération *substitute* à E et aux identifiants k et l. Soient ρ et ρ' les dénotations abstraites de E, respectivement, avant et après cette application de l'opération *substitute*. Finalement, l'opération *normalize* se réapplique elle-même à la collection de structures E. Comme le nombre d'identifiants utilisés par E a diminué d'une unité par rapport à la collection de départ, on peut affirmer, par hypothèse d'induction, que l'exécution complète de l'opération *normalize* transforme la collection E de telle sorte que sa dénotation abstraite soit égale à $\sim_{\rho'}$.

Mais le lemme 10 nous permet d'affirmer que $\sim_{\rho'} = \sim_{\rho}$. Il s'en suit que la dénotation abstraite finale de E est bien égale à $\sim_{\rho}$. □

2.3.2.4 L'opération *unify*

Cette dernière opération permet de "résoudre des équations entre termes clos" (voir le paragraphe 2.3.3).

Définition 25 (Définition opérationnelle de l'opération *unify*)**.** Soit E, une collection de structures (bien formée). Soient i et j, deux identifiants d'ensembles de structures utilisés par E. L'opération *unify* appliquée à E, i et j, applique d'abord l'opération *substitute* à E, i et j (ou à E, j et i). Ensuite, elle applique l'opération *normalize* à E.

Théorème 20 (Spécification de l'opération *unify*)**.** *Avec les notations de la définition 25, soit ρ, la dénotation abstraite de la collection de structures E, avant l'exécution de l'opération unify. Soit encore $(T_i)_{i\in I}$, la dénotation concrète de E (avant).*

1. *Après l'exécution de l'opération, la collection de structures E dénote abstraitement la plus petite congruence $\sim$ telle que l'on ait (pour tous les termes s, t) :*

$$\begin{aligned} s\ \rho\ t &\implies s\ \sim\ t \\ s \in T_i \text{ et } t \in T_j &\implies s\ \sim\ t \end{aligned}$$

2. *Soient $T = dom(\rho)$ et $\tilde{T} = dom(\sim)$. On a :*

$$\tilde{T} \ = \ \{\ t \in \mathcal{T} \mid \exists s \in T : s\ \sim\ t\ \}$$

Preuve du théorème 20. Conséquence immédiate des théorèmes 18 et 19. □

2.3.3 Résolution d'équations entre termes clos

Notre notion de collection de structures permet de résoudre de manière particulièrement simple et élégante un problème de décision classique (voir [Koz77, Sho78, NO80, BM07]) que nous énonçons ci-dessous. Nous montrerons plus tard que notre implémentation concrète des collections de structures rend aussi notre méthode très efficace.

Soit Eq un ensemble fini d'équations entre termes clos que l'on peut écrire, par exemple :

$$Eq = \{s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m\}\ (m \geq 0)$$

Le problème classique est de décider si deux termes quelconques s et t sont égaux dans tous les modèles des équations de Eq. Pour résoudre ce problème, nous utilisons l'algorithme suivant.

1. Créer une collection de structures E contenant l'unique structure $null : i_{null}$.
2. Pour chaque équation $u = v$ de Eq,
   (a) Appliquer l'opération *toSet* à E et u, puis à E (éventuellement modifiée) et à v.
   (b) Soient i et j, les identifiants renvoyés par ces deux applications de l'opération. Appliquer l'opération *unify* à E, i et j.
3. Appliquer l'opération *toSet* à E et à s, puis à t. Les termes s et t sont égaux dans tous les modèles des équations si et seulement si les identifiants renvoyés par ces deux applications de l'opération *toSet* sont égaux.

Remarque 2.

1. On fera attention au fait que, dans l'algorithme ci-dessus, le symbole E représente un "objet global" qui est susceptible d'être modifié par chaque application d'une opération. (Il en sera de même dans l'implémentation concrète, décrite dans la section 2.4.)
2. On notera aussi que le symbole d'égalité utilisé ci-dessus pour écrire les équations est "purement formel" et que le fait d'écrire l'équation $s = t$ ne signifie pas qu'on affirme que les termes s et t sont identiques. D'autres auteurs préfèrent utiliser un autre symbole et écrire, par exemple : $s \approx t$.

Pour démontrer la correction de l'algorithme, nous donnons d'abord une nouvelle définition.

Définition 26 (Congruence sur tous les termes définie par un ensemble d'équations)**.** Soit Eq, un ensemble d'équations entre termes clos. Nous appelons *congruence définie par* Eq, la plus petite congruence $\cong_{Eq}$ sur tous les termes telle que l'on ait, quels que soient les termes s et t :

$$s = t \in Eq \implies s \cong_{Eq} t$$

Il est clair qu'on a le théorème :

Théorème 21. *Soient Eq et Eq', deux ensembles d'équations entre termes clos. On a :*

$$Eq' \subseteq Eq \implies \cong_{Eq'} \subseteq \cong_{Eq}$$

On restreint ensuite la relation $\cong_{Eq}$ comme suit.

Définition 27 (Congruence associée aux sous-termes d'un ensemble d'équations)**.** Soit Eq, un ensemble d'équations entre termes clos. Soit T l'ensemble de tous les sous-termes des termes figurant dans les équations de Eq. Soit aussi $\tilde{T}$ l'ensemble de termes défini par l'égalité :

$$\tilde{T} \quad = \quad \{\ t \in \mathcal{T} \mid \exists s \in T : s \ \cong_{Eq} \ t\ \}$$

On note $\sim_{Eq}$ la relation entre termes définie comme suit :

$$\sim_{Eq} \quad = \quad \{\ \langle s, t\rangle \mid s, t \in \tilde{T} \ \text{ et } \ s \ \cong_{Eq} t\ \}$$

La correction de l'algorithme proposé repose sur le théorème suivant.

Théorème 22 (Résolution d'un ensemble d'équations entre termes clos)**.** *Après exécution de l'étape* 2 *de l'algorithme présenté à la page 80, la collection de structures E a pour dénotation abstraite la relation $\sim_{Eq}$. (Si Eq est vide, on convient que l'ensemble de ses sous-termes est égal à $\{null\}$).*

La preuve du théorème 22 utilise le lemme que voici :

Lemme 11. *Soit Eq, un ensemble d'équations entre termes clos. Soit ρ, la plus petite relation réflexive dont le domaine est égal à l'ensemble des sous-termes des termes figurant dans les équations de Eq et telle que*

$$s = t \in Eq \quad \Longrightarrow \quad s\,\rho\,t.$$

On a :

$$\sim_{Eq} \quad = \quad \sim_{\rho}\,.$$

Preuve du lemme 11. Il est clair que l'ensemble d'équations Eq et la relation ρ imposent exactement les mêmes contraintes d'égalités sur les termes. Par conséquent, les relations $\cong_{Eq}$ et $\cong_{\rho}$ sont égales. Donc, les relations $\sim_{Eq}$ et $\sim_{\rho}$ le sont aussi, en vertu du théorème 10. □

Nous pouvons maintenant prouver le théorème proprement dit.

Preuve du théorème 22. Nous démontrons le théorème par récurrence sur le nombre d'équations de l'ensemble Eq. Si l'ensemble Eq est vide, le résultat est facile à vérifier. Sinon, soit $u = v$, l'équation traitée en dernier lieu par l'étape 2 de l'algorithme et soit Eq' l'ensemble des autres équations. Soient ρ et ρ', les relations entre termes correspondant à Eq et Eq', selon l'énoncé du lemme 11. Notons U, l'ensemble des sous-termes

de u et v. Notons T', l'ensemble des sous-termes de Eq' et T l'ensemble de ceux de Eq. On a, évidemment : $T = T' \cup U$. On peut supposer, par hypothèse d'induction, qu'avant de traiter la dernière équation, la collection de structures E dénote abstraitement la relation $\sim_{\rho'}$. Soit $(S_k)_{k\in K}$, la dénotation concrète de E avant la dernière exécution de l'opération *unify*. Soit $\sim'$ sa dénotation abstraite. On a :

1. $dom(\sim') \;=\; \{\; t \in \mathcal{T} \mid \exists s \in T : s \;\cong_{\rho'}\; t \;\}$
2. $\sim' \;=\; \{\; \langle s,t\rangle \mid s,t \in \; dom(\sim') \;\text{ et }\; s \;\cong_{\rho'} t \;\}$
3. $u \in S_i \;\text{ et }\; v \in S_j$, où i et j sont définis comme indiqués au point 2.b) de l'algorithme.

Soit maintenant $\sim$, la dénotation de E après la dernière exécution de l'opération *unify*. Nous devons montrer que $\sim \;=\; \sim_{Eq}$. Pour cela, il suffit de montrer que $\sim \;=\; \sim_\rho$, d'après le lemme 11.

1. On a : $\sim_\rho \;\subseteq\; \sim$.
   Pour le voir, il suffit, d'après les théorèmes 3 et 4, de montrer que $\rho \subseteq \sim$. Soient s et t, deux termes tels que $s\,\rho\,t$. Il y a trois cas à envisager.
   (a) Les termes s et t sont identiques (réflexivité de ρ). Il est dès lors évident que l'on a $s \sim t$, puisque $s,t \in T$, $T \subseteq \; dom(\sim)$ et puisque la relation $\sim$ est réflexive.
   (b) L'équation $s = t$ appartient à Eq'. Alors, on a $s\,\rho'\,t$. D'où, successivement, $s \;\cong_{\rho'}\; t$ (définition de $\cong_{\rho'}$), puis $s \;\sim'\; t$ (définition de $\sim'$, ci-dessus, et $s,t \in T$) et, enfin, $s \sim t$ (par la spécification de l'opération *unify*).
   (c) On a $s = u$ et $t = v$. Dans ce cas, $s \in S_i$ et $t \in S_j$, comme précisé au point 3., ci-dessus. Il s'en suit que $s \sim t$, par la spécification de l'opération *unify*.

2. D'autre part : $\sim \;\subseteq\; \sim_\rho$.
   Pour le prouver, nous allons utiliser le fait que $dom(\sim') \subseteq dom(\sim_\rho)$. Cela découle des trois faits suivants :
   (a) $dom(\sim') \;=\; \{t \in \mathcal{T} \mid \exists s \in T : s \cong_{\rho'} t\}$,
   (b) $dom(\sim_\rho) \;=\; \{t \in \mathcal{T} \mid \exists s \in T : s \cong_\rho t\}$,
   (c) $\cong_{\rho'} \;\subseteq\; \cong_\rho \quad (\text{car } \rho' \subseteq \rho)$.
   Il suffit maintenant de montrer que la relation $\sim_\rho$ respecte toutes les conditions imposées à la relation $\sim$ par la spécification de *unify* sauf, peut-être, la minimalité. Il y a donc deux points à vérifier. Soient deux termes s et t.

(a) On a :

$$s \sim' t \implies s \sim_\rho t.$$

En effet, $s \sim' t$ implique d'abord qu'on a : $s \cong_{\rho'} t$ et $s, t \in dom(\sim')$. D'où, il vient : $s \cong_\rho t$ et $s, t \in dom(\sim_\rho)$. D'où enfin : $s \sim_\rho t$ (par le théorème 10).

(b) On a, de plus :

$$s \in S_i \text{ et } t \in S_j \implies s \sim_\rho t.$$

Car : $s \in S_i$ et $t \in S_j$ impliquent qu'on a $s \cong_{\rho'} u$ et $v \cong_{\rho'} t$. D'où : $s \cong_\rho u$ et $v \cong_\rho t$. D'où, encore : $s \cong_\rho t$, puisque $u \cong_\rho v$. Comme $s, t \in dom(\sim_\rho)$, on a bien : $s \sim_\rho t$, par le même théorème 10. □

Nous prouvons maintenant la correction de l'algorithme complet.

Preuve de la correction de l'algorithme de la page 80. Après application de l'opération *toSet* aux termes s et t, à l'étape 3. de l'algorithme, la collection de structures E dénote abstraitement la congruence $\sim$ telle que

1. $dom(\sim) = \{ v \in \mathcal{T} \mid \exists u \in T : u \cong_{Eq} v \}$
2. $\sim = \{ \langle u, v \rangle \mid u, v \in dom(\sim) \text{ et } u \cong_{Eq} v \}$

où T est l'ensemble de tous les sous-termes des termes figurant dans les équations de Eq augmenté des sous-termes de s et t. Comme s et t appartiennent à $dom(\sim)$, on a $s \cong_{Eq} t$ si et seulement si $s \sim t$, c'est-à-dire si et seulement si les identifiants renvoyés par les deux dernières applications de l'opération *toSet* sont égaux (puisque la collection de structures finale E est normalisée et que sa dénotation concrète est une présentation de la relation entre termes $\sim$). □

Exemple 7. Nous donnons d'abord un exemple pour illustrer les deux premières étapes de l'algorithme de la page 80, c'est-à-dire la résolution d'un ensemble d'équations entre termes clos. Considérons les deux équations suivantes :

$$b = f(f(f(a))) \qquad a = f(f(a)).$$

Nous appliquons d'abord l'opération *toSet* aux termes figurant en parties gauche et droite de la première équation. Nous obtenons une collection de structures E^0 telle que la suivante :

$E^0_1 = \{b : 1\}$ $E^0_2 = \{a : 2\}$ $E^0_3 = \{f(2) : 3\}$ $E^0_4 = \{f(3) : 4\}$ $E^0_5 = \{f(4) : 5\}$.

Cette collection de structures dénote la famille d'ensembles de termes, ci-dessous :

$$T_1^0 = \{b\} \quad T_2^0 = \{a\} \quad T_3^0 = \{f(a)\} \quad T_4^0 = \{f(f(a))\} \quad T_5^0 = \{f(f(f(a)))\}.$$

Pour résoudre l'équation $b = f(f(f(a)))$, nous appliquons l'opération *unify* à 1, 5 et E^0, ce qui nous donne E^1 :

$$E_1^1 = \{b : 1,\ \ f(4) : 1\} \quad E_2^1 = \{a : 2\} \quad E_3^1 = \{f(2) : 3\} \quad E_4^1 = \{f(3) : 4\}.$$

On peut noter que, dans ce cas, la première application de l'opération *substitute* produit directement une collection de structures normalisée, de sorte que l'application de l'opération *normalize* qui la suit ne modifie rien. La famille d'ensembles de termes correspondante est :

$$T_1^1 = \{b,\ \ f(f(f(a)))\} \qquad T_2^1 = \{a\} \qquad T_3^1 = \{f(a)\} \qquad T_4^1 = \{f(f(a))\}.$$

Pour résoudre la seconde équation, nous appliquons d'abord l'opération *toSet* aux termes a et $f(f(a))$. Aucune de ces deux applications ne modifie la collection de structures courante puisque $a \in T_2^1$ et $f(f(a)) \in T_4^1$. Pour résoudre la seconde équation, nous appliquons donc l'opération *unify* à 2, 4 et E^1. L'opération *substitute* est d'abord appliquée aux mêmes arguments et produit la collection de structures E^2 :

$$E_1^2 = \{b : 1,\ \ f(2) : 1\} \qquad E_2^2 = \{a : 2,\ \ f(3) : 2\} \qquad E_3^2 = \{f(2) : 3\}.$$

Cette collection de structures dénote une famille de trois ensembles infinis de termes :

$$T_1^2 = \{b,\ f(a),\ f(f(f(a))),\ f(f(f(f(f(a))))),\ f(f(f(f(f(f(f(a))))))),\ \ldots\}$$

$$T_2^2 = \{a,\ f(f(a)),\ f(f(f(f(a)))),\ f(f(f(f(f(f(a)))))),\ \ldots\}$$

$$T_3^2 = \{f(a),\ f(f(f(a))),\ f(f(f(f(f(a))))),\ f(f(f(f(f(f(f(a))))))),\ \ldots\}.$$

On constate que les ensembles E_1^2 et E_3^2 contiennent deux structures possédant la même clé. Par conséquent, l'opération *substitute* est appliquée une deuxième fois, à 1, 3 et E^2, ce qui nous donne E^3 :

$$E_1^3 = \{b : 1,\ \ f(2) : 1\} \qquad\qquad E_2^3 = \{a : 2,\ \ f(1) : 2\}.$$

La collection E^3 est le résultat final. Elle dénote la famille de deux ensembles infinis de termes, ci-dessous.

$$T_1^3 = \{b,\ f(a),\ f(f(b)),\ f(f(f(a))),\ f(f(f(f(b)))),\ f(f(f(f(f(a))))),\ \ldots\}$$

$$T_2^3 = \{a,\ f(b),\ f(f(a)),\ f(f(f(b))),\ f(f(f(f(a)))),\ f(f(f(f(f(b))))),\ \ldots\}.$$

On vérifie facilement que les ensembles T_1^3 et T_2^3 contiennent la totalité des termes égaux à b et a dans tous les modèles possibles des deux

équations considérées. Ils contiennent aussi tous les termes égaux à $f(a)$, $f(f(a))$, ou $f(f(f(a)))$, c'est-à-dire aux autres sous-termes des termes figurant dans les équations. Ces constatations sont en parfait accord avec l'énoncé du théorème 22.

Exemple 8. Nous montrons maintenant que l'algorithme complet de la page 80 permet de déterminer si deux termes sont égaux dans tous les modèles des équations considérées à l'exemple 7.

1. Considérons d'abord les deux termes suivants :

   $$g(a, f(a)) \qquad\qquad g(f(b), b).$$

   L'application de l'opération *toSet* au premier terme $g(a, f(a))$ et à la collection de structures E^3 de l'exemple 7, nous donne la collection de structures E^4 ci-dessous :

   $$E_1^4 = \{b : 1,\ f(2) : 1\} \qquad E_2^4 = \{a : 2,\ f(1) : 2\} \qquad E_3^4 = \{g(2,1) : 3\}.$$

   Ensuite, l'application de la même opération *toSet* au second terme $g(f(b), b)$ ne modifie pas la collection de structures. Elle renvoie l'identifiant 3, tout comme la première application de l'opération. Les deux termes sont donc égaux dans tous les modèles des deux équations.

2. Considérons ensuite ces deux autres termes :

   $$g(a, f(a)) \qquad\qquad g(b, f(b)).$$

   L'application de l'opération *toSet* au premier terme $g(a, f(a))$ et à la collection de structures E^3 de l'exemple 7, nous donne, comme dans l'exemple précédent, la collection de structures E^4 ci-dessous :

   $$E_1^4 = \{b : 1,\ f(2) : 1\} \qquad E_2^4 = \{a : 2,\ f(1) : 2\} \qquad E_3^4 = \{g(2,1) : 3\}.$$

   L'opération *toSet* est ensuite appliquée au second terme $g(b, f(b))$. Cette fois, elle modifie la collection de structures, comme suit :

   $$E_1^5 = \{b : 1,\ f(2) : 1\} \qquad E_2^5 = \{a : 2,\ f(1) : 2\}$$

   $$E_3^5 = \{g(2,1) : 3\} \qquad E_4^5 = \{g(1,2) : 4\}.$$

   Elle renvoie l'identifiant 4, alors que la première application de l'opération a renvoyé 3. Les deux termes ne sont donc pas égaux dans tous les modèles des deux équations : la collection de structures E^5 constitue, en fait, un modèle partiel des deux équations, dans lequel les deux termes dénotent des valeurs différentes (respectivement 3 et 4). Ce point est développé plus bas (paragraphe 2.3.3.1, point 3.).

2.3.3.1 Discussion

Nous concluons cette sous-section par quelques réflexions plus générales.

1. L'algorithme de décision que nous venons de proposer ne procède pas tout à fait comme nous l'avions annoncé à la page 62. Il est nettement plus simple, grâce à l'utilisation de l'opération *toSet* qui renvoie les identifiants des ensembles de termes contenant les termes à comparer. Le seul inconvénient de cette méthode c'est qu'elle nécessite d'éventuellement étendre la collection de structures pour que sa dénotation contienne les termes à comparer. On peut imaginer des situations dans lesquelles ces extensions seraient trop "gourmandes en mémoire". Nous laissons au lecteur intéressé le soin d'écrire un algorithme utilisant nos collections de structures pour implémenter la méthode de la page 62.
2. Notre utilisation de la terminologie "résolution d'équations entre termes clos" mérite sans doute une justification car, quand on parle d'équations, on a généralement à l'esprit des équations entre des expressions ou des termes contenant des variables et le but de la résolution est de donner aux variables des valeurs qui rendent les expressions ou les termes égaux. Dans notre cas, nous ne donnons pas des valeurs à des variables figurant dans les termes mais nous fournissons une représentation très explicite de *tous* les termes égaux à *tous* les sous-termes figurant dans les équations, ce qui peut être vu, nous semble-t-il, comme une forme de résolution d'équations. En particulier, les constantes figurant dans les termes reçoivent des valeurs, de manière similaire aux variables des équations "classiques" (voir l'exemple 8).

   Pour essayer de bien comprendre la différence entre ces deux utilisations du terme "résolution d'équations", on peut observer que dans l'utilisation "classique" du mot, on fixe une interprétation déterminée, unique, des termes (par exemple, des nombres) et on cherche à résoudre les équations par rapport à cette interprétation. Dans notre cas, on cherche à déterminer toutes les interprétations qui rendent les équations vraies. Ainsi, on peut mettre en parallèle notre méthode de résolution d'équations avec l'algorithme d'unification des termes en logique du premier ordre. Dans ce cas, l'interprétation des termes est unique et fixée : c'est l'égalité "syntaxique" des termes. Une équation telle que $a = f(a)$ n'a pas de solution car l'interprétation syntaxique des termes n'est pas un modèle de cette équation. Si x est une variable, l'équation $x = f(x)$ n'a pas de solution car les termes t et $f(t)$ sont syntaxiquement

différents, quel que soit le terme t. Néanmoins, dans certaines versions de Prolog, on accepte l'existence de termes infinis. Dans ce cas, l'équation à une solution : le terme constitué d'une infinité d'occurrences du symbole f. Il est intéressant de constater que la représentation habituelle (en mémoire) d'un tel terme infini est semblable à la structure $f(1) : 1$, dans notre terminologie. Ainsi, même si le sens donné aux équations n'est pas le même, les représentations utilisées et les résultats des algorithmes présentent d'intéressantes similitudes.

3. La collection de structures résultant de l'application des deux premières étapes de l'algorithme de la page 80 est un modèle partiel des équations Eq dans un sens précis, expliqué ci-dessous. La dénotation abstraite finale de la collection de structures E est égale à la congruence $\sim_{Eq}$, dont l'extension à tous les termes est égale à la relation $\cong_{Eq}$. L'ensemble $\mathcal{T}_{/\cong_{Eq}}$ des classes d'équivalences de la relation $\cong_{Eq}$ est un modèle des équations Eq si on interprète chaque symbole de fonction f par la fonction $\hat{f}$ telle que, quels que soient les termes t_1 et t_2, on a : $\hat{f}([t_1], [t_2]) = [f(t_1, t_2)]$, où $[t]$ désigne la classe d'équivalence du terme t pour la relation $\cong_{Eq}$. On construit un modèle équivalent (isomorphe) à celui-là, en associant à chaque classe d'équivalence T de $\cong_{Eq}$ un identifiant $i_T \in \mathcal{I}$ et en interprétant chaque symbole de fonction f par la fonction $\hat{f}$ telle que $\hat{f}(i_{[t_1]}, i_{[t_2]}) = i_{[f(t_1,t_2)]}$. Soit $(T_j)_{j \in J}$, la dénotation concrète de E, après résolution des équations de Eq. Chaque ensemble T_j est une classe d'équivalence de la relation $\cong_{Eq}$. On peut donc choisir les identifiants de ce second modèle de telle sorte que $i_{T_j} = j$ (pour chaque $j \in J$). Pour chaque structure $f(j_1, j_2) : j$ de E, on a, évidemment : $i_{T_j} = \hat{f}(i_{T_{j_1}}, i_{T_{j_2}})$, par définition de la dénotation concrète de E. Donc, on a : $j = \hat{f}(j_1, j_2)$, c'est-à-dire que la collection de structures E donne exactement les valeurs des fonctions $\hat{f}$, dans le second modèle, chaque fois que leurs arguments et leur valeur (résultat) appartiennent à J. Et il y a plus : le reste du modèle peut être construit "à la demande", en appliquant l'opération *toSet* à autant de termes que l'on veut. L'information fournie par E est donc équivalente à une description complète du modèle (pour autant qu'on lui demande seulement de permettre le calcul exact et rapide des fonctions $\hat{f}$).

   Le modèle obtenu ci-dessus est aussi le plus général possible : si deux termes sont égaux dans ce modèle, ils le sont dans n'importe quel modèle.

4. Le problème de décision considéré dans cette sous-section 2.3.3

est classique [Koz77, Sho78, NO80, BM07] mais notre méthode pour le résoudre est nouvelle et présente plusieurs avantages sur les solutions qui lui ont été apportées par d'autres auteurs. Nous discutons cette question à la section 2.6.

2.4 Implémentation des collections de structures

Nous décrivons maintenant les principes d'une implémentation efficace de la structure de données présentée "à un haut niveau" dans la section 2.3. L'efficacité de cette implémentation est étudiée d'un point de vue "théorique" dans la première partie de la section 2.5 et illustrée expérimentalement dans la seconde partie de cette section. Pour donner plus de poids à cette étude expérimentale, nous présentons aussi, dans la section 2.6, une comparaison d'efficacité avec certaines méthodes proposées précédemment par d'autres auteurs.

Nous découpons cette section en quatre parties. La sous-section 2.4.1 présente une liste d'objectifs *a priori* suffisants pour garantir l'optimalité d'une implémentation des collections de structures. La sous-section 2.4.2 définit une représentation des collections de structures par un ensemble de structures de données "logiques". La sous-section 2.4.3 explique comment réaliser une implémentation des opérations de la sous-section 2.3.2 au moyen de ces structures de données logiques. On y justifie que les objectifs d'efficacité fixés dans la sous-section 2.4.1 sont atteints par cette implémentation. Enfin, la sous-section 2.4.4 donne quelques explications sur la réalisation effective des structures de données "logiques" dans un langage de programmation "classique" (nous avons utilisé Java).

2.4.1 Objectifs d'efficacité

L'implémentation des collections de structures que nous proposons a été conçue par rapport aux objectifs d'efficacité énoncés et justifiés ci-dessous.

1. Il est souhaitable que l'opération *toSet*, appliquée à une collection de structures E et à un terme t, s'effectue en $O(|\,t\,|)$ où $|\,t\,|$ désigne la taille (nombre de symboles) du terme t. Une meilleure complexité est impossible à atteindre : il faut au moins parcourir le terme en entier pour construire sa représentation.
2. L'opération *substitute*, appliquée à une collection de structures E et à deux identifiants d'ensembles de structures k et l, doit idéalement s'effectuer en $O(|\,S_l\,|)$ où S_l est l'ensemble des structures de E contenant au moins une occurrence de l et $|\,S_l\,|$ est le nombre d'éléments de cet ensemble [4]. Pour cela, on doit pouvoir "parcourir" l'ensemble S_l en un temps de l'ordre de $O(|\,S_l\,|)$ et effectuer, dans chaque structure, un remplacement de l'identifiant

4. L'ensemble S_l est, en général, différent de E_l, on a : $E_l \subseteq S_l$.

l en $O(1)$ (voir la notation 3). Pour réaliser ce dernier point, il suffit de savoir retirer la structure originale en $O(1)$ et ajouter la structure modifiée en $O(1)$.

3. Pour que l'opération *normalize* s'effectue optimalement, il est souhaitable de savoir déterminer en $O(1)$ une paire de structures de même clé, lorsqu'au moins une telle paire de structures existe (ou d'établir en $O(1)$ qu'il n'en existe pas).

2.4.2 Structures de données "logiques"

Nous donnons maintenant une description "logique" des structures de données que nous utilisons pour réaliser les objectifs fixés, ci-dessus. Nous caractérisons ces structures logiques en termes d'opérations associées et de leur complexité théorique. La sous-section 2.4.4 explique comment elles sont réalisées concrètement, dans notre implémentation en Java.

1. Nos algorithmes font grand usage de valeurs appelées *identifiants*. Outre les identifiants d'ensembles de structures, introduits par la définition 14, nous associons un *identifiant de clé* à chaque clé $f(i_1, i_2)$ d'une collection de structures E et un *identifiant de structure* à chaque structure $f(i_1, i_2) : i$, de E. Nous utilisons le symbole *ikey* pour dénoter un identifiant de clé et le symbole *istruct* pour dénoter un identifiant de structure.
2. Nous maintenons une structure de données appelée *table des clés* pour représenter la bijection existant entre les clés de la collection de structures E et leurs identifiants. Cette table permet d'obtenir l'identifiant d'une clé, à partir de celle-ci, en $O(1)$ et réciproquement. Il est également possible de lui ajouter ou de lui retirer un couple formé d'une clé et d'un identifiant de clé, en $O(1)$. (Ceci nécessite de pouvoir choisir un "nouvel identifiant" en $O(1)$.)
3. Nous maintenons une structure de données similaire pour représenter la bijection existant entre les paires de la forme $\langle ikey, i\rangle$, où *ikey* est l'identifiant d'une clé $f(i_1, i_2)$, et l'identifiant *istruct* de la structure $f(i_1, i_2) : i$. Nous l'appelons *table des structures*.
4. Pour chaque identifiant de clé *ikey*, nous maintenons une liste contenant les identifiants de toutes les structures ayant pour clé celle dont *ikey* est l'identifiant. Nous notons cette liste *sameKey*(*ikey*). Elle est implémentée de telle sorte qu'on puisse lui retirer ou lui ajouter un identifiant de structures en $O(1)$. On peut la parcourir en $O(n)$ où n est le nombre de ses éléments.

5. Pour chaque identifiant d'ensemble de structures i, nous maintenons trois listes notées $sameId_1(i)$, $sameId_2(i)$ et $sameId(i)$ contenant, respectivement, les identifiants de toutes les structures d'une des formes $f(i, i_2) : i_3$, $f(i_1, i) : i_3$, $f(i_1, i_2) : i$, selon le cas (avec f, i_1, i_2, i_3, quelconques). Les opérations sur ces listes ont la même complexité que pour les listes du point précédent.
6. Nous maintenons une liste unique, nommée *dBList*, contenant tous les identifiants de clés correspondant aux clés pour lesquelles plusieurs structures (ayant cette clé) existent. Il est possible de choisir un élément dans cette liste en $O(1)$. Il est aussi possible de lui ajouter et de lui retirer un élément quelconque en $O(1)$.

2.4.3 Implémentation des opérations sur les collections de structures

Nous pouvons maintenant décrire en détails les algorithmes implémentant les opérations sur les collections structures, définies à la sous-section 2.3.2.

2.4.3.1 Algorithme de l'opération *toSet*

L'algorithme procède récursivement exactement comme expliqué à la définition 22. Toutefois, chaque ajout d'une structure $f(i_1, i_2) : i$ à la collection de structures E nécessite le choix de trois nouveaux identifiants i, *ikey* et *istruct*, une mise à jour de la table des clés et de la table des structures, et l'ajout de l'identifiant *istruct* aux listes $sameKey(ikey)$, $sameId_1(i_1)$, $sameId_2(i_2)$ et $sameId(i)$. Vu les exigences imposées aux opérations ces objets, il est clair que la complexité de l'algorithme est de l'ordre de $O(|\,t\,|)$.

2.4.3.2 Algorithme de l'opération *substitute*

Nous utilisons les notations de la définition 23.

Nous parcourons l'une après l'autre chacune des listes $sameId_1(l)$, $sameId_2(l)$ et $sameId(l)$. Pour chacun des identifiants de structures *istruct* rencontré, nous effectuons les opérations suivantes.

1. Nous retirons l'identifiant de structures *istruct* de toutes les listes dans lesquelles il figure. Cela est réalisé comme suit.
   (a) On détermine la paire $\langle ikey, i\rangle$, correspondant à *istruct*, dans la table des structures.

(b) On détermine la clé $f(i_1, i_2)$ correspondant à *ikey*, dans la table des clés.

(c) On retire *istruct* des listes $sameId_1(i_1)$, $sameId_2(i_2)$, $sameId(i)$ et $sameKey(ikey)$.

2. Nous retirons la paire $\langle ikey, i \rangle$, définie au point précédent, de la table des structures.
3. Si la liste $sameKey(ikey)$ ne contenait précédemment qu'un seul élément, nous retirons de la table des clés le couple formé de $f(i_1, i_2)$ et *ikey*. Par contre, si elle en contenait exactement deux, nous retirons *ikey* de la liste *dBList*.
4. Nous remplaçons l par k dans la structure $f(i_1, i_2) : i$. Notons $f(i'_1, i'_2) : i'$, le résultat de ce remplacement.
5. Nous ajoutons, si elle n'y figurait pas déjà, la structure $f(i'_1, i'_2) : i'$ à la représentation de la collection de structures en effectuant les points suivants.

   (a) Si la clé $f(i'_1, i'_2)$ ne figure pas dans la table des clés, on lui associe un nouvel identifiant de clé et on place la paire obtenue dans la table des clés. Dans tous les cas, on note $ikey'$, l'identifiant de clé associé à $f(i'_1, i'_2)$.

   (b) Si la paire $\langle ikey', i' \rangle$ ne figure pas dans la table des structures, on lui associe un nouvel identifiant de structures et on place la paire obtenue dans la table. Dans tous les cas, on note $istruct'$, l'identifiant de structures associé à $\langle ikey', i' \rangle$.

   (c) On ajoute l'identifiant $istruct'$ à chacune des listes $sameId_1(i'_1)$, $sameId_2(i'_2)$, $sameId(i')$ et $sameKey(ikey')$, si il n'y figurait pas déjà.

   (d) Si la liste $sameKey(ikey')$ contenait précédemment un seul élément, différent de $istruct'$, on ajoute $ikey'$ dans la liste *dBList*.

La correction de l'algorithme proposé ci-dessus ne peut être vérifiée qu'en l'exécutant "symboliquement" pas à pas et en constatant qu'après chaque considération d'un identifiant de structures *istruct*, associé à une structure $f(i_1, i_2) : i$, il maintient les propriétés exigées des différentes structures de données manipulées par l'algorithme (voir la sous-section 2.4.2).

En ce qui concerne la complexité théorique de l'algorithme (voir la sous-section 2.4.1, point 2), nous pouvons la justifier comme suit.

1. Soit S_l, l'ensemble des structures contenant l'identifiant l, avant exécution de l'opération *substitute*. Chaque structure de S_l n'est considérée qu'une seule fois par l'algorithme puisque son identifiant est immédiatement retiré de toutes les listes dans lesquelles il figure (point 1. de l'algorithme).
2. Le nombre d'opérations effectuées par l'algorithme pour traiter chaque structure est fini et borné (il n'y a aucune boucle mentionnée dans les différents points de l'algorithme). De plus, toutes ces opérations s'effectuent en $O(1)$, d'après les hypothèses faites sur les structures de données utilisées. Le traitement de chaque structure est donc réalisé en $O(1)$.

2.4.3.3 Algorithme de l'opération *normalize*

L'algorithme réalisant l'opération *normalize* procède exactement comme décrit dans la définition 2.3.2.3 en utilisant la liste *dBList* pour tester si la collection de structures est normalisée et pour sélectionner une paire de structures de même clé, sinon. En effet, la collection de structures est normalisée si et seulement si la liste *dBList* est vide. Si elle ne l'est pas, son premier élément fournit un identificateur de clé *ikey* correspondant à au moins deux structures. Les identifiants de deux telles structures sont directement obtenus en retirant de la liste *sameKey*(*ikey*) ses deux premiers éléments. L'ensemble de toutes ces opérations s'effectue en $O(1)$ comme requis à la sous-section 2.4.1, point 3.

Néanmoins, les considérations précédentes ne nous renseignent pas beaucoup sur la complexité théorique globale de l'opération *normalize*. Celle-ci fait l'objet, pour l'essentiel, de la section 2.5, mais nous pouvons déjà faire la remarque suivante : le choix de deux identifiants k et l correspondant à deux structures de même clé donne deux possibilités d'application de l'opération *substitute*. Soit on remplace l par k, soit k par l. Il semble *a priori* évident qu'il vaut mieux choisir de remplacer l'identifiant qui figure dans le plus petit nombre de structures. Pour cela, il est nécessaire d'ajouter et de maintenir à jour, pour chaque identifiant d'ensemble de structures i, un compteur du nombre d'éléments de l'ensemble S_i de toutes les structures contenant l'identifiant i.

L'opération *unify* se composant d'une application de *substitute* suivie d'une application de *normalize*, son algorithme procède de la même façon et ne nécessite pas d'être commenté davantage ici.

2.4.4 Notes sur l'implémentation concrète

Nous donnons maintenant des informations sur la manière dont les structures "logiques" de la sous-section 2.4.2 sont réalisées concrètement dans notre implémentation en Java.

1. Nous utilisons des nombres entiers non négatifs (valeurs du type `int`) comme identifiants. Nous fixons une valeur maximale à chaque sorte d'identifiant. Ces valeurs sont notées ci-dessous `NBRSETS`, `NBRKEYS` et `NBRSTRUCTS`[5].

2. Nous avons conçu une "nouvelle" structure de données (classe Java), nommée `DoubleList`, capable de représenter *simultanément* un nombre quelconque d'ensembles *disjoints* d'entiers non négatifs inférieurs à une borne supérieure N. Ces ensembles sont tous représentés à l'intérieur de deux tableaux d'entiers indicés de 0 à $N-1$, notés `pred` et `succ`. Ces deux tableaux permettent de représenter les différents ensembles sous forme de listes doublement chaînées utilisant les éléments des tableaux dont les indices sont les éléments des ensembles. L'ordre des éléments dans les listes est indéterminé : il dépend, en fait, de l'ordre dans lequel les éléments ont été ajoutés à l'ensemble. Si la valeur v n'a pas de prédécesseur (de successeur) dans l'ensemble, on a : $\texttt{pred}[v] = -1$ ($\texttt{succ}[v] = -1$).

   Les éléments des deux tableaux dont les indices ne figurent dans aucun des ensembles représentés, sont regroupés en une seule liste simplement chaînée, utilisant un des deux tableaux. Pour distinguer ces positions des positions "utilisées" par les ensembles représentés, on "code" les indices de cette liste par des nombres négatifs ≤ -2. Cette liste des positions libres permet de choisir en $O(1)$ une valeur ne figurant dans aucun ensemble.

   Il est clair qu'avec cette représentation, on peut aussi enlever un élément d'un ensemble, et le remettre dans la liste des positions libres en $O(1)$ et, également, ajouter à un ensemble un nouvel élément, choisi dans la liste des positions libres, en $O(1)$.

3. Sur base de la classe `DoubleList`, nous avons construit une deuxième classe, nommée `MultiList`, paramétrée par un second entier positif M, qui maintient exactement M ensembles disjoints de valeurs (comprises entre 0 et $N-1$) et fournit un accès au premier élément de chacun de ces ensembles, grâce à l'utilisation d'un troisième tableau indicé de 0 à $M-1$.

5. De façon plus correcte, ces symboles représentent le nombre total d'identifiants disponibles dans chaque catégorie d'identifiants.

4. La totalité des listes *sameKey*(*ikey*), $sameId_1(i)$, $sameId_2(i)$ et *sameId*(i) est représentée par quatre "instances" de la classe `MultiList`. Pour la première de ces instances, on choisit $M =$ `NBRKEYS` et $N =$ `NBRSTRUCTS`. Pour les trois autres, on choisit $M =$ `NBRSETS` et $N =$ `NBRSTRUCTS`. La liste libre de l'instance contenant les listes *sameId*(i) est utilisée pour choisir un nouvel identifiant de structure. En effet, chaque structure appartient à un ensemble de structures unique. Donc, les identifiants de structures libres sont ceux qui n'appartiennent à aucune liste *sameId*(i).
5. Sur base de la classe `DoubleList`, nous avons construit une troisième classe, nommée `OneList`, qui maintient un seul ensemble de valeurs ainsi qu'une liste complémentaire de valeurs libres.
6. Une instance de la classe `OneList` maintient la liste des identificateurs d'ensembles de structures E_i. Sa liste libre sert à choisir un nouvel identificateur i, chaque fois qu'un nouvel ensemble de structures est créé par l'opération *toSet*. Lorsqu'un identificateur l est retiré de l'ensemble des identificateurs d'ensembles de structures, par l'opération *substitute*, il est rajouté à la liste libre de cette instance pour être réutilisé plus tard (comme un identifiant "tout neuf"). Cette instance est évidemment telle que $N =$ `NBRSETS`.
7. Une autre instance de la classe `OneList` contient la liste *dBList* des identificateurs de toutes les clés correspondant à deux structures au moins. Pour cette instance, on a, bien sûr, $N =$ `NBRKEYS`.
8. Pour représenter la table des clés et celle des structures, nous utilisons deux tables de hachage. La table de hachage réalisant la table des clés est un simple tableau indicé par les identificateurs de clé et accédé selon la méthode de l'adressage ouvert. Par contre, la table des structures est réalisée, d'une part, par un tableau, indicé de 0 à `NBRSTRUCTS` $- 1$, contenant les paires $\langle ikey, i \rangle$ et, d'autre part, par une cinquième instance de la classe `MultiList`, telle que $N =$ `NBRSTRUCTS`. (M peut être choisi indépendamment de `NBRSETS`, `NBRKEYS` et `NBRSTRUCTS`). La fonction de hachage doit garantir que les identifiants de structures sont répartis équitablement entre les M listes maintenues par l'instance. (On peut noter que cette seconde table de hachage est superflue lorsque la collection de structures E est normalisée ou "quasi normalisée", car on peut alors lui substituer l'instance de la classe `MultiList` représentant les listes *sameKey*(*ikey*). Malheureusement, il arrive dans certaines applications que la collection E contienne un très grand nombre de structures de même clé. La seconde table de hachage est alors indispensable pour maintenir les performances annoncées

pour les algorithmes. Nous en montrerons un exemple à la section 2.5.)

2.5 Complexité théorique et étude expérimentale d'efficacité

Nous avons établi que le temps d'exécution d'une application utilisant les collections de structures est proportionnel au nombre de renommages effectués par les opérations *substitute* et à la somme des tailles des termes "ajoutés" à la collection de structures au cours du traitement. Nous nous attelons maintenant à estimer la complexité globale d'une telle application. Nous allons le faire en combinant étude théorique et expérimentation pratique.

Nous choisissons pour objectif de donner au lecteur des informations générales valables pour n'importe quelle application quitte à fournir plus tard, dans de futurs travaux, des caractérisations plus précises pour telle ou telle application particulière. Dès lors, nous nous concentrerons d'abord sur l'étude de complexité d'un traitement concernant à effectuer une suite d'opérations *substitute* ou *unify* sur une collection préexistante, sans y ajouter de nouveaux termes en cours de route (sections 2.5.1 à 2.5.5). Nous compléterons cela ensuite par l'étude de complexité de la résolution d'un ensemble d'équations entre termes clos, en distinguant deux options : créer tous les termes préalablement ou les créer au fur et à mesure de la résolution des équations (section 2.5.6).

Introduisons maintenant la première étude : effectuer une suite d'opérations *substitute* ou *unify* sur une collection préexistante. Il y a trois paramètres à prendre en compte pour une telle étude : le nombre M d'identifiants de la collection de départ, le nombre N de structures de cette collection et le nombre m d'opérations *substitute* effectuées. Nous cherchons d'abord des résultats de complexité en fonction de ces trois paramètres, puis nous détaillons plus particulièrement le cas où $m = M - 1$ qui correspond au "pire des cas" dans lequel tous les identifiants sont finalement unifiés.

Il s'avère qu'il est éclairant de distinguer trois "sortes" de collection de structures correspondant à des "sortes" de traitements ou à des "phases" successives d'un même traitement [6] :

1. Nous appelons collection *creuse* (au sens strict), une collection normalisée dans laquelle le nombre de structures N est égal au nombre d'identifiants M. Une telle collection représente un ensemble de termes tous "non équivalents" deux à deux, c'est-à-dire que chaque ensemble de structures E_i contient une seule struc-

6. Ces distinctions peuvent aussi s'appliquer "localement" à une partie d'une collection.

ture. Une telle collection apparaît toujours au début d'un traitement si on crée tous les termes nécessaires au traitement avant de les mettre en relation.

2. Nous appelons collection *dense* (au sens strict), une collection normalisée dans laquelle il existe exactement une structure $f(i_1, i_2) : i$ pour chaque couple d'identifiants i_1, i_2 utilisés par la collection (et chaque symbole de fonction f, utilisé par la collection). Cela implique que chaque terme, constructible avec les symboles de fonction utilisés par la collection, est représenté dans la collection (par un identifiant déterminé). Autrement dit, la collection représente une théorie équationnelle complète.
3. Nous appelons collection *hyper dense* (au sens strict), une collection dans laquelle il existe exactement une structure $f(i_1, i_2) : i$ pour chaque triplet d'identifiants i_1, i_2, i utilisés par la collection, et chaque symbole de fonction f. Une telle collection n'est évidemment pas normalisée en général.

Nous utilisons aussi les adjectifs "creuse", "dense" et "hyper dense" pour des collections qui sont "proches" de collections "creuse", "dense" et "hyper dense" au sens strict. Il est clair que plus une collection est "proche" d'une collection au sens strict, plus elle se comportera comme elle vis-à-vis des algorithmes qui s'y appliquent. Typiquement, dans une collection creuse, les ensembles de structures E_i contiennent peu de structures, mais beaucoup, dans une collection dense. Dans une collection dense, le nombre N de structures est de l'ordre de M^2. Dans une collection hyper dense, il peut être de l'ordre de M^3. Une collection hyper dense "n'a pas d'intérêt pour elle-même" puisque seules les collections normalisées sont stables vis-à-vis de l'opération *normalize* mais, comme nous le verrons expérimentalement, il arrive très souvent que des collections hyper denses soient créés de manière intermittente avant d'être réduites à une collection normalisée (dense).

2.5.1 Bornes supérieures au nombre de renommages

Nous cherchons d'abord des formules pour caractériser le pire des cas d'exécution d'un algorithme qui effectue m opérations *substitute* sur les identifiants d'une collection de structures comportant, au départ, M identifiants et N structures ($0 \leq m < M \leq N$). Nous ne faisons ici aucune hypothèse particulière sur la forme de la collection ni sur la façon dont les identifiants, auxquels on applique l'opération *substitute*, sont choisis. Nous établissons plusieurs bornes supérieures en commençant par une borne précise mais paramétrée par des nombres dépendants

de chaque exécution particulière. Nous en tirons d'autres bornes, moins précises mais utilisant des paramètres plus généraux, jusqu'à trouver des expressions ne dépendant que de M, N et m.

Théorème 23. *Soit $Tren_m$, le nombre de renommages effectués par les m premières applications de l'opération substitute aux identifiants d'une collection de structures E. Nous notons*

- *I : l'ensemble des M identifiants utilisés par E (au départ),*
- *pour chaque $i \in I$, n_i : le nombre d'occurrences de l'identifiant i dans les structures de E,*
- *$n = \sum_{i \in I} n_i$: le nombre total d'occurrences d'identifiants dans les structures de E,*
- *I^m : l'ensemble des identifiants de E (modifiée) après la $m-$ème application de substitute. Cet ensemble comporte $M - m$ identifiants.*
- *pour chaque $i \in I^m$, I_i^m : l'ensemble des identifiants de la collection de départ qui ont été renommés en i par les m applications de substitute. On a :*

$$I = \bigcup_{i \in I^m} I_i^m \tag{2.1}$$

- *pour chaque $i \in I^m$, n_i^m : le nombre d'occurrences, dans la collection de départ, des identifiants qui ont été renommés en i par les m applications de substitute. On a :*

$$n_i^m = \sum_{j \in I_i^m} n_j \tag{2.2}$$

Tout ça étant fixé, on a :

$$Tren_m \leq \frac{1}{2}\left(\sum_{i \in I^m} n_i^m \log_2 n_i^m - \sum_{i \in I} n_i \log_2 n_i\right) \tag{2.3}$$

Preuve du théorème 23. Nous démontrons le théorème par récurrence sur m. Le résultat est "évident" si $m = 0$ car $I^0 = I$ et $n_i^0 = n_i$ pour tout $i \in I$. Et aussi, bien sûr, parce que $Tren_0 = 0$.

Supposons maintenant que $0 < m < M$ et que le résultat soit vrai pour $m - 1$. Soient k et l les identifiants auxquels la $m-$ème opération *substitute* est appliquée. Nous supposons que l'on renomme l'identifiant qui a le moins d'occurrences dans la collection (à ce moment-là). Nous supposons, de plus, qu'il s'agit de l. Soit r, le nombre de renommages effectués. On a, nécessairement :

$$r \leq \min\left(n_k^{m-1}, n_l^{m-1}\right) \tag{2.4}$$

En effet, puisque l'opération *substitute* ne crée jamais de nouvelle structure mais ne fait que les renommer, le nombre d'occurrences de k et l est nécessairement inférieur ou égal, respectivement à n_k^{m-1} et à n_l^{m-1}. De plus, r est inférieur ou égal au nombre d'occurrences de l et au nombre d'occurrences de k, dans la collection, à ce moment. Donc r est inférieur ou égal à n_k^{m-1} et à n_l^{m-1}. De l'inégalité 2.4, on tire immédiatement :

$$r + \frac{1}{2}\left(n_k^{m-1}\log_2 n_k^{m-1} + n_l^{m-1}\log_2 n_l^{m-1}\right) \leq \frac{1}{2}n_k^m \log_2 n_k^m \qquad (2.5)$$

en utilisant le fait que $n_k^m = n_k^{m-1} + n_l^{m-1}$ et une propriété prouvée à l'annexe B (Théorème 32).

Nous pouvons maintenant prouver la relation 2.3, en utilisant l'hypothèse de récurrence et le fait que $n_i^m = n_i^{m-1}$ si $i \in I^m \setminus \{k\}$:

$$\begin{aligned}
Tren_m &= r + Tren_{m-1} \\
&\leq r + \frac{1}{2}\Big(\sum_{i\in I^{m-1}} n_i^{m-1}\log_2 n_i^{m-1} - \sum_{i\in I} n_i \log_2 n_i\Big) \\
&= r + \frac{1}{2}(n_k^{m-1}\log_2 n_k^{m-1} + n_l^{m-1}\log_2 n_l^{m-1}) \\
&\quad + \frac{1}{2}\Big(\sum_{i\in I^{m-1}\setminus\{k,l\}} n_i^{m-1}\log_2 n_i^{m-1} - \sum_{i\in I} n_i \log_2 n_i\Big) \\
&\leq \frac{1}{2}n_k^m \log_2 n_k^m + \frac{1}{2}\Big(\sum_{i\in I^m\setminus\{k\}} n_i^m \log_2 n_i^m - \sum_{i\in I} n_i \log_2 n_i\Big) \\
&= \frac{1}{2}\Big(\sum_{i\in I^m} n_i^m \log_2 n_i^m - \sum_{i\in I} n_i \log_2 n_i\Big)
\end{aligned}$$

□

Note 11. L'intuition derrière la relation 2.3 est la suivante. Si la collection de départ était telle que $n_i = 1$ pour tout $i \in I$, la relation 2.3 deviendrait simplement :

$$Tren_m \leq \frac{1}{2}\sum_{i\in I^m} n_i^m \log_2 n_i^m \qquad (2.6)$$

Une telle collection n'est pas bien formée, en général, puisque certains (beaucoup) de ses ensembles E_i sont vides. On peut cependant créer de telles collections et leur appliquer librement l'opération *substitute*. En réalité, la collection de départ du théorème 23 peut être construite par applications successives de l'opération *substitute* à une telle collection mal formée. Le nombre de renommages nécessaires sera borné par le nombre

$$\frac{1}{2}\sum_{i\in I} n_i \log_2 n_i \qquad (2.7)$$

En soustrayant le nombre 2.7 du terme de droite de la relation 2.6, on retrouve la relation 2.3. Intuitivement, en français, on calcule la borne supérieure en supposant qu'on est parti d'une collection avec $n_i = 1$ pour tout $i \in I$ et on lui retire le nombre de renommages nécessaires (au pire) pour construire à partir de cette collection, la "vraie" collection de départ.

Nous calculons maintenant des versions simplifiées, moins précises mais plus synthétiques, de la borne supérieure fournie par le théorème 23.

Théorème 24. *Nous reprenons les notations du théorème 23 et nous y ajoutons les suivantes. Pour chaque $i \in I^m$, nous notons m_i^m, le nombre d'identifiants de l'ensemble I_i^m, moins 1, c'est-à-dire le nombre d'applications de l'opération substitute ayant été exécutées pour remplacer les occurrences de ces identifiants par des occurrences de i. On a donc : $m = \sum_{i \in I^m} m_i^m$. Avec ces notations, on a les résultats suivants.*

$$Tren_m \;\leq\; \frac{1}{2} \sum_{i \in I^m} n_i^m \log_2(m_i^m + 1) \tag{2.8}$$

$$Tren_m \;\leq\; \frac{3}{2}\, N\, \log_2(m+1) \tag{2.9}$$

$$Tren_m \;\leq\; \frac{3}{2}\, N\, \log_2 M \tag{2.10}$$

Preuve du théorème 24. Nous démontrons d'abord que pour chaque $i \in I_m$, on a :

$$n_i^m \log_2 n_i^m - \sum_{j \in I_i^m} n_j \log_2 n_j \;\leq\; n_i^m \log_2(m_i^m + 1) \tag{2.11}$$

Pour le voir, remarquons que l'on a

$$\sum_{j \in I_i^m} n_j \;=\; n_i^m$$

et que, pour n'importe quelle combinaison de valeurs n_j vérifiant cette égalité, l'expression

$$\sum_{j \in I_i^m} n_j \log_2 n_j$$

prend sa plus petite valeur lorsque toutes les valeurs n_j sont égales, et, donc, égales à

$$\frac{n_i^m}{m_i^m + 1}.$$

Cette valeur minimale est

$$\begin{aligned}\sum_{j\in I_i^m} n_j \log_2 \frac{n_i^m}{m_i^m+1} &= (m_i^m+1)\frac{n_i^m}{m_i^m+1}\log_2\frac{n_i^m}{m_i^m+1}\\ &= n_i^m(\log_2 n_i^m - \log_2(m_i^m+1))\\ &= n_i^m \log_2 n_i^m - n_i^m \log_2(m_i^m+1)\end{aligned}$$

Ce résultat est prouvé en annexe (Théorème 33).
Dès lors,

$$\begin{aligned}& n_i^m \log_2 n_i^m - \sum_{j\in I_i^m} n_j \log_2 n_j\\ \leq\ & n_i^m \log_2 n_i^m - (n_i^m \log_2 n_i^m - n_i^m \log_2(m_i^m+1))\\ =\ & n_i^m \log_2(m_i^m+1)\end{aligned}$$

Nous pouvons maintenant démontrer la relation 2.8 à partir de la relation 2.3, en utilisant la relation 2.11 :

$$\begin{aligned}Tren_m &\leq \frac{1}{2}(\sum_{i\in I^m} n_i^m \log_2 n_i^m - \sum_{i\in I} n_i \log_2 n_i)\\ &= \frac{1}{2}(\sum_{i\in I^m} n_i^m \log_2 n_i^m - \sum_{i\in I^m}\sum_{j\in I_i^m} n_j \log_2 n_j)\\ &= \frac{1}{2}\sum_{i\in I^m}(n_i^m \log_2 n_i^m - \sum_{j\in I_i^m} n_j \log_2 n_j)\\ &\leq \frac{1}{2}\sum_{i\in I^m} n_i^m \log_2(m_i^m+1)\end{aligned}$$

Nous prouvons maintenant la relation 2.9 à partir de la relation 2.8, en observant que chaque nombre m_i^m est borné par m et que $\sum_{i\in I^m} n_i^m$ est borné par $3N$:

$$\begin{aligned}Tren_m &\leq \frac{1}{2}\sum_{i\in I^m} n_i^m \log_2(m_i^m+1)\\ &\leq \frac{1}{2}\sum_{i\in I^m}(n_i^m \log_2(m+1))\\ &= \frac{1}{2}(\sum_{i\in I^m} n_i^m)\log_2(m+1)\\ &\leq \frac{3}{2}N\log_2(m+1)\end{aligned}$$

Finalement, la relation 2.10 est obtenue "trivialement" à partir de la relation 2.9, en faisant $m = M-1$. Cette relation fournit une borne supérieure absolue au nombre de renommages effectués par n'importe quelle

application effectuant des opérations *substitute*, *unify* ou *normalize* sur une collection de structures contenant au départ M identifiants et N structures, sans lui ajouter de nouvelles structures, bien entendu. □

Note 12. Nous avons établi les résultats de cette section en choisissant, lors d'une application d'une opération *substitute* à deux identifiants k et l, de remplacer celui qui comporte les moins d'occurrences dans les structures de la collection. D'autres critères de choix sont envisageables. Par exemple, on peut choisir l'identifiant qui figure dans le moins de structures. Avec cette méthode, on obtient les mêmes résultats que ci-dessus mais la preuve du théorème 23 est un peu moins simple. On peut aussi penser choisir de remplacer l si $m_l^m \leq m_k^m$ (et k, sinon). On obtient alors, par un raisonnement direct, la relation de complexité :

$$Tren_m \leq \sum_{i \in I^m} n_i^m \log_2(m_i^m + 1) \tag{2.12}$$

qui est moins bonne que la relation 2.8 et, *a fortiori*, que la relation 2.3. Cette façon de faire semble donc un peu moins efficace, conformément à "l'intuition". On peut encore choisir l dont la "hauteur", dans "l'arbre des fusions" des identifiants de I_k^m et de I_l^m, est la plus petite. Cette méthode de choix équivaut à la précédente.

Pour finir, on notera que toutes ces méthodes de choix sont aisées à mettre en œuvre en associant à chaque identifiant de la collection le compteur adéquat.

2.5.2 Étude expérimentale

Nous complétons l'étude théorique de la section 2.5.1 par une étude expérimentale. L'étude de la section 2.5.1 se base sur le nombre de structures présentes dans la collection initiale et ne prend pas en compte le fait que ce nombre peut diminuer lors des renommages de structures. Il est même inévitable qu'il diminue lorsque la valeur de m devient proche de M. Indépendamment de cela, on peut prédire que le nombre de renommages moyen est moindre que ce qui est prévu par notre étude des pires des cas. Nous proposons donc maintenant une étude expérimentale de ce nombre de renommages moyen.

Nous menons les expérimentations suivantes. Nous créons "aléatoirement" des collections de structures normalisées utilisant $M = 1.000$ identifiants et un nombre de structures N égal à 1.000, 10.000, 100.000 ou 1.000.000. Si $M = 1.000$, la collection est creuse au sens strict. Si $M = 1.000.000$, elle est dense au sens strict. Les collections sont construites comme suit. Nous construisons, d'abord, une collection creuse contenant

1.000 identifiants et 1.000 structures, en choisissant, pour chaque nouvelle structure $f(i_1, i_2) : i$, un nouvel identifiant i et deux identifiants i_1 et i_2 déjà utilisés dans des structures précédemment créées. Pour la première structure créée, on choisit $i_1 = i_2 = i_{null}$. Toutes les structures utilisent le même symbole de fonction f. On complète ensuite la collection en choisissant "aléatoirement" les structures complémentaires nécessaires, en veillant à ne jamais choisir deux structures possédant la même clé. Nous exécutons ensuite $M - 1$ opérations *substitute* sur chacune de ces collections jusqu'à obtenir une collection qui ne contient plus qu'un identifiant et deux structures. Nous le faisons selon deux "stratégies" bien différentes. Dans le premier cas, chaque opération *substitute* est appliquée à deux identifiants k et l choisis "au hasard" dans l'ensemble des identifiants restants (encore utilisés par la collection). Dans le second cas, nous exécutons des opérations *unify*, au lieu d'opérations *substitute*. Il s'en suit que les identifiants ne sont plus toujours tirés "au hasard" : lorsque les opérations *substitute* sont appliquées à l'intérieur d'une opération *normalize* (déclenchée par une opération *unify*), les identifiants choisis correspondent à la présence dans la collection de deux structures possédant la même clé. Pour chacune de ces stratégies, nous collectons trois informations : le nombre N_m de structures encore présentes dans la collection, le nombre $Tren_m$ de renommages effectués depuis le début du traitement et Ren_m, le nombre "instantané" de renommages effectués à l'étape m. Le nombre m représente le nombre d'opérations *substitute* déjà effectuées depuis le début du traitement, qu'elles soient ou non effectuées à l'intérieur d'une opération *normalize*. Ces nombres diffèrent évidemment en fonction du contenu de la collection de départ et des choix aléatoires d'identifiants. Les nombres Ren_m, en particulier, sont extrêmement variables d'une exécution à une autre. Pour obtenir une estimation du comportement moyen des algorithmes, nous répétons les mesures un grand nombre de fois et nous prenons des moyennes. Les expériences sont effectuées 10.000 fois si $N = 1.000$, 5.000 fois si $N = 10.000$, 2.500 fois si $N = 100.000$, et 1.250 fois si $N = 1.000.000$.

Nous montrons maintenant les graphiques correspondant à ces expérimentations en y ajoutant des éléments d'interprétation. Sur les trois graphiques correspondants aux trois types de grandeurs N_m, $Tren_m$ et Ren_m, les courbes tracées en continu et notées (a), (b), (c), (d), correspondent à l'exécution purement "aléatoire" d'opérations *substitute* tandis que les courbes tracées en traits interrompus et notées (A), (B), (C), (D) correspondent à l'exécution d'opérations *unify*. L'ordre alphabétique croissant des lettres correspond aux valeurs croissantes du nombre initial de structures : $N = 1.000$, 10.000, 100.000, 1.000.000.

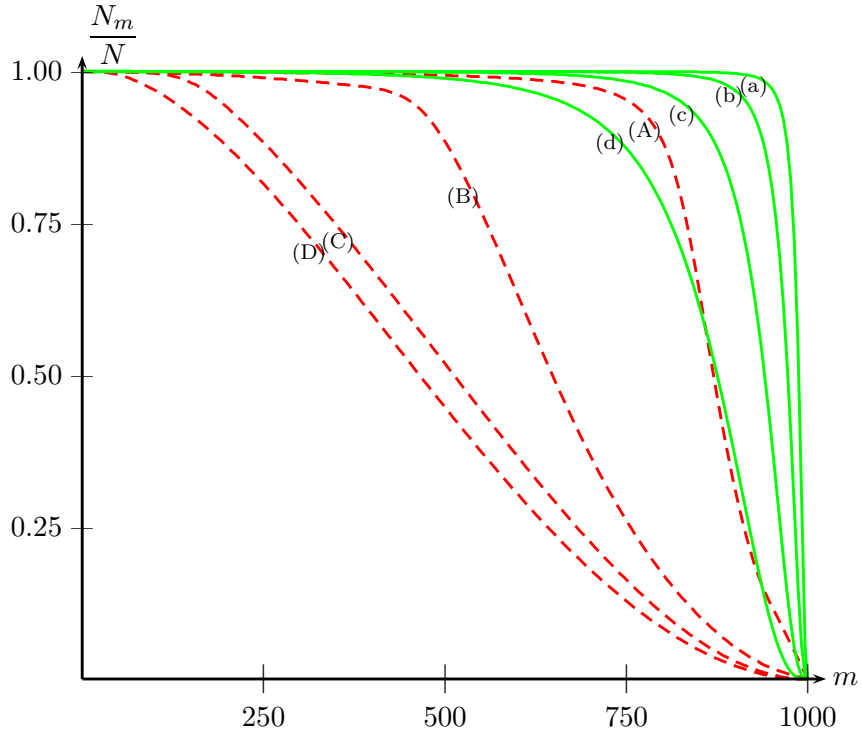


Figure 2.1 – Nombre de structures N_m, en fonction de m

L'évolution du nombre N_m de structures est présenté à la figure 2.1. Les courbes présentées sur la figure décrivent en fait l'évolution de la quantité $\frac{N_m}{N}$. Cela permet de réunir toutes les courbes en une seule figure et facilite la comparaison des résultats. L'algorithme se comporte assez différemment selon la stratégie utilisée. Si l'on exécute des opérations *substitute* sur des identifiants choisis purement "au hasard", le nombre de structures ne diminue pratiquement pas jusqu'à ce que N soit proche de $(M-m)^3$. À ce moment-là, toutes les structures possibles sont utilisées et les opérations *substitute* qui suivent ne peuvent que supprimer toutes les structures contenant l'identifiant qui est renommé, ce qui explique la diminution extrêmement rapide de la valeur N_m. La chute rapide de N_m se produit d'autant plus tard que N est petit puisqu'il faut évidemment une plus grande valeur de m pour que N soit proche de $(M-m)^3$ si la collection est creuse que si elle est dense. Dans le cas où l'on exécute des opérations *unify*, la décroissance de N_m se produit plus tôt et elle est comparativement moins brusque. L'explication principale en est que l'opération *unify* supprime une structure de la collection de structures dès que celle-ci contient deux structures possédant la même clé, ce que l'autre stratégie néglige de faire. Cela se passe assez tard si la collection

est creuse mais très tôt, dès le début, si la collection est dense.

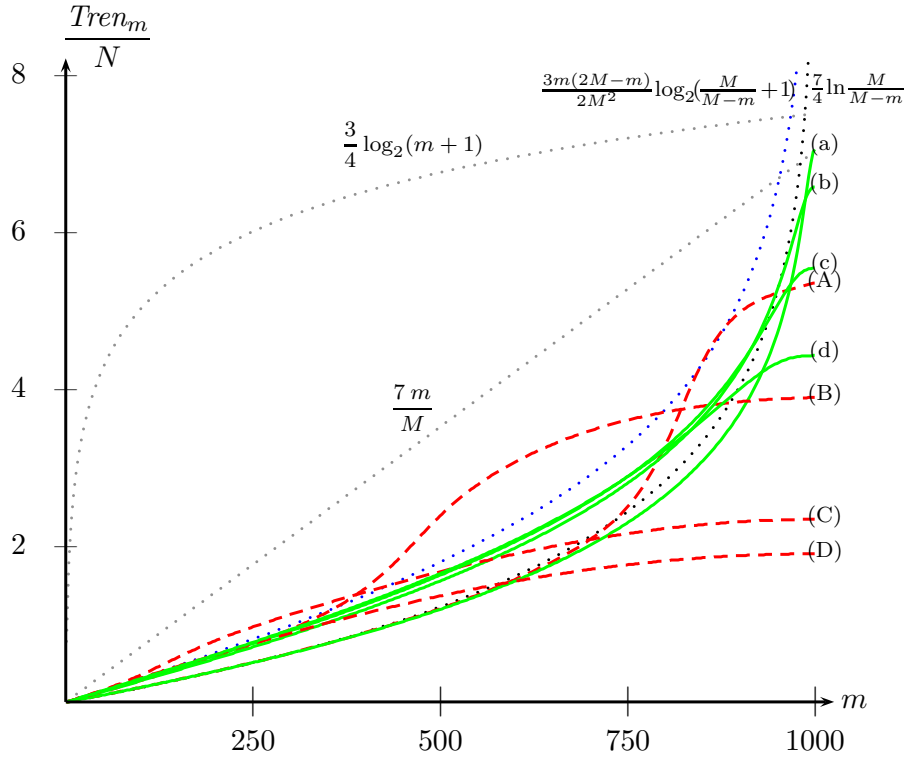


Figure 2.2 – Nombre total de renommages $Tren_m$, en fonction de m

La figure 2.2 fournit l'information la plus pertinente pour estimer l'efficacité pratique de nos algorithmes, en moyenne : le nombre total de renommages effectués après m opérations *substitute*. À nouveau, pour obtenir une figure lisible et synthétique, nous donnons des courbes non pour la valeur $Tren_m$ mais pour la valeur $\frac{Tren_m}{N}$. On constate sur la figure que les résultats expérimentaux sont nettement meilleurs que les bornes supérieures calculées dans la section 2.5.1 : dans tous les cas, on a : $Tren_m \leq \frac{3}{4} N \log_2(M)$ et, si l'on applique *unify* : $Tren_m < \frac{5.5\,N}{M} m < 5.5\,N$. On voit que pour des valeurs de m très proches de M, l'utilisation de l'opération *unify* provoque moins de renommages que celle de l'utilisation "aléatoire" de *substitute*. Néanmoins, pour des valeurs petites et moyennes de m, l'utilisation de *substitute* provoque moins de renommages et ce nombre grandit relativement lentement, de l'ordre de $N \log_2 \frac{M}{M-m}$.
La forme des courbes de la figure 2.2 et la différence de comportement des algorithmes selon l'utilisation de *substitute* ou de *unify* s'explique

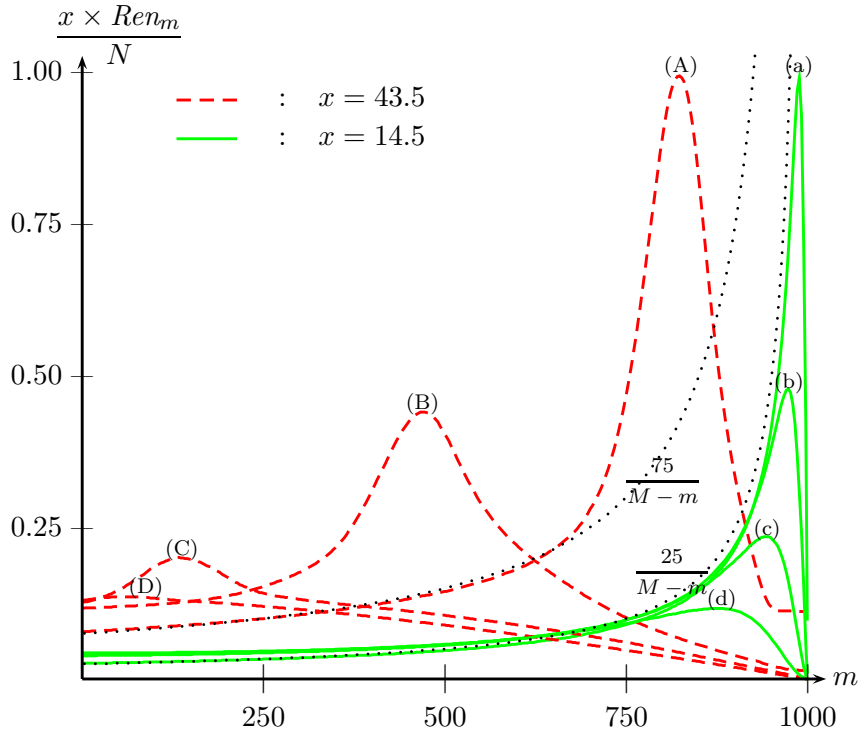


Figure 2.3 – Nombre instantané de renommages Ren_m, en fonction de m

mieux en considérant la figure 2.3 qui présente les nombres de renommages "instantanés", c'est-à-dire les courbes dérivées, par rapport à m, de celles de la figure 2.2. Dans le cas de l'opération *substitute*, la valeur de Ren_m est approximativement égale à $\frac{2\,N}{M-m}$ jusqu'à ce qu'on ait, à peu près : $N = (M-m)^3$, c'est-à-dire : $m = M - \sqrt[3]{N}$; soit, pour les valeurs choisies pour M et N : $m = 990$ (a), 978.5 (b), 953.5 (c), 900 (d). À ce stade, le nombre de renommages diminue extrêmement rapidement puisque pratiquement toutes les structures renommées sont supprimées. Il est de l'ordre de $(M-m)^2$. Dans le cas de l'opération *unify*, l'explication des courbes de la figure 2.3 est plus difficile. Nous pensons qu'elle peut se formuler comme suit. Dans un premier temps, tant que peu de structures ayant des clés égales sont créées par renommage, les algorithmes se comportent comme pour l'opération *substitute*. Lorsque la collection devient dense, le nombre de structures se met à décroître mais le nombre "instantané" de renommages augmente parce que les identifiants qui ont déjà participé à un renommage (ou plus) ont un plus grand nombre d'occurrences et figurent donc plus souvent que les autres en position d'identifiant principal des structures (i dans la structure $f(i_1, i_2) : i$). Dès lors, ces identifiants sont plus souvent renommés. Il se

produit donc une phase intermédiaire durant laquelle les identifiants qui n'ont jamais participé à un renommage sont ignorés et les autres identifiants sont répétitivement renommés jusqu'au moment où il n'en existe plus que quelques-uns, voire un seul. Cet identifiant survivant est finalement unifié avec ceux qui étaient ignorés jusque-là. Ceux-ci sont bien entendus "absorbés" par l'identifiant dominant, moyennant un faible taux de renommage, correspondant au plus au nombre d'occurrences de ces identifiants dans la collection de départ. Les maximums (A), (B), (C), (D), correspondent au moment où la partie de la collection contenant les identifiants "actifs" (non ignorés) devient pratiquement hyper dense. Les structures qui les contiennent sont alors rapidement supprimées par renommage (voir la figure 2.1). Finalement, les identifiants ignorés jusque-là sont absorbés de manière uniforme et le nombre de renommages est assez semblable à ce qu'il était au début du traitement. On le voit particulièrement bien sur la courbe (A) de la figure 2.3. Pour les grandes valeurs de N (courbes (C) et (D)), correspondant à des collections denses, au départ, la phase menant à la détermination d'un identifiant "dominant" se produit au début du traitement et le nombre de renommages décroît ensuite lentement.

2.5.3 Estimation du nombre de renommages en moyenne

L'étude expérimentale de la section 2.5.2 montre que le comportement en moyenne des algorithmes exécutant répétitivement des opérations *substitute* ou *unify*, sur une collection de structures donnée initialement, est significativement meilleur que ce qui est prédit par les bornes supérieures établies à la section 2.5.1, particulièrement quand la collection de départ est dense. Nous proposons, dans cette section-ci, des estimations mathématiques de ce comportement moyen. Une étude tout à fait rigoureuse, basée sur les probabilités et l'analyse combinatoire, en est, nous semble-t-il, très difficile. Nos estimations sont donc basée sur des modélisations simplifiées dont la pertinence sera évaluée par une comparaison avec l'étude expérimentale, à la section 2.5.4.

Comme pour l'étude expérimentale, nous distinguons le cas, plus simple, où l'opération *substitute* est appliquée répétitivement à des paires d'identifiants choisis "au hasard" (sous-section 2.5.3.1), du cas, plus intéressant mais plus difficile, ou c'est l'opération *unify* qui est appliquée répétitivement (sous-section 2.5.3.2).

2.5.3.1 Paires d'identifiants choisies au hasard

Nous proposons d'abord deux estimations du comportement moyen de l'algorithme qui, comme dans la section 2.5.1, ne prennent pas en compte le fait que le nombre de structures de la collection diminue nécessairement à partir d'un nombre d'étapes qui varie selon que la collection soit plutôt creuse (à la fin) ou plutôt dense (dès le début). L'estimation la plus simple découle du raisonnement suivant. Si le nombre N de structures reste constant et si les identifiants sont répartis de manière à peu près uniforme dans les structures, le nombre d'occurrences d'un identifiant, après m étapes, est à peu près égal à

$$\frac{3\,N}{(M-m)}.$$

Ce nombre est aussi à peu près le nombre de renommages qui seront effectués à l'étape $m+1$, car il est rare qu'un même identifiant figure plus d'une fois dans une structure. Le nombre total de renommages effectués lors des m premières applications de l'opération *substitute* est donc, approximativement égal à

$$\sum_{i=0}^{m-1} \frac{3\,N}{(M-m)}.$$

Cette somme peut être approximée par l'intégrale suivante :

$$\int_0^m \frac{3\,N}{(M-x)}\,dx.$$

Cette intégrale se calcule aisément et nous fournit donc l'estimation :

$$Tren_m \;\approx\; 3\,N\,\ln\frac{M}{M-m}. \tag{2.13}$$

On constate que cette estimation est proche de (légèrement supérieure à) la borne supérieure 2.10, lorsque $m = M-1$. En général, cependant, elle fournit une meilleure approximation que la borne 2.9, comme on peut le déduire de la figure 2.2.

La seconde estimation que nous proposons est basée sur l'inégalité 2.8. Notons p^m le nombre d'identifiants de I^m ayant participé à au moins une des m premières exécutions de l'opération *substitute*[7]. C'est le nombre des ensembles I_i^m qui contiennent plus d'un identifiant. Une étude expérimentale montre que l'on a, en moyenne :

$$p^m \;\approx\; \frac{m\,(M-m)}{M} \tag{2.14}$$

7. Nous utilisons les notations du théorème 24.

et que cette approximation est précise. Notons J^m l'ensemble des identifiants de I^m ayant participé à au moins une application de l'opération *substitute*. Pour les identifiants de $I^m \setminus J^m$, on a :

$$\log_2(m_i^n + 1) \;=\; 0.$$

La borne supérieure donnée par la partie droite de l'inégalité 2.8 est donc égale à l'expression

$$\frac{1}{2} \sum_{i \in J^m} n_i^m \log_2(m_i^m + 1). \tag{2.15}$$

Nous pouvons estimer la valeur de cette expression en multipliant une estimation de la somme

$$\sum_{i \in J^m} n_i^m, \tag{2.16}$$

par une estimation du logarithme de la moyenne

$$\frac{\sum_{i \in J^m}(m_i^m + 1)}{p^m}. \tag{2.17}$$

La somme 2.16 est égale au nombre d'occurrences des identifiants de $I \setminus J^m$ dans la collection de structures de départ, c'est-à-dire, en moyenne, en supposant que tous les identifiants ont la même fréquence dans la collection initiale :

$$3\frac{N}{M}(m+p^m) \;\approx\; 3\frac{N}{M}\left(m + \frac{m\,(M-m)}{M}\right) \;=\; 3\frac{m\,(2M-m)\,N}{M^2}, \tag{2.18}$$

en tenant compte de l'estimation 2.14.

La moyenne 2.17 se simplifie comme suit, en tenant aussi compte de l'estimation 2.14 :

$$\frac{\sum_{i \in J^m} m_i^m + 1}{p^m} \;=\; \frac{m + p^m}{p^m} \;=\; \frac{m}{p^m} + 1 \;\approx\; \frac{m}{\frac{m\,(M-m)}{M}} + 1 \;=\; \frac{M}{M-m} + 1. \tag{2.19}$$

On obtient finalement l'estimation :

$$\begin{aligned} \mathit{Tren}_m \;&\approx\; \frac{1}{2}\left(3\,\frac{m\,(2M-m)\,N}{M^2}\right)\log_2\left(\frac{M}{M-m}+1\right) \\ &=\; \frac{3}{2}\,\frac{m\,(2M-m)\,N}{M^2}\,\log_2\left(\frac{M}{M-m}+1\right). \end{aligned} \tag{2.20}$$

La courbe correspondant à cette estimation est représentée à la figure 2.2. On y voit que l'estimation fournit une borne supérieure pour Tren_m qui

est précise sauf pour les grandes valeurs de m, lorsque les nombres N_m décroissent rapidement. On peut vérifier que si $m = M - 1$, l'estimation 2.20 est presqu'exactement égale à la borne supérieure 2.10. Au contraire, si m est petit par rapport à M, l'estimation 2.20 est à peu près égale à

$$3\,\frac{N}{M}\,m,$$

ce qui correspond à l'intuition : le nombre de renommages effectués est égal au nombre moyen d'occurrences d'un identifiant, multiplié par le nombre d'opérations *substitute* déjà effectuées.

Nous passons maintenant à l'étape suivante, c'est-à-dire : établir des estimations des nombres Tren_m qui tiennent compte de la décroissance des nombres de structures N_m. Nous le faisons en raffinant la méthode utilisée pour établir l'estimation 2.13. (Il n'est pas possible de raffiner 2.20, qui se base sur une borne supérieure établie sans tenir compte de la décroissance des nombres de structures.) Nous postulons que l'on a :

$$N_m \;\approx\; \frac{N\;(M-m)^3}{N\;+\;(M-m)^3} \tag{2.21}$$

Cette estimation résulte de la combinaison des deux estimations suivantes.

1. Les N structures de la collection de départ sont, après m applications de l'opération *substitute*, transformées en N_m structures n'utilisant plus que $M-m$ identifiants. Il y a $(M-m)^3$ telles structures [8]. Nous postulons que la valeur moyenne de N_m est égale au nombre moyen de structures distinctes obtenues en tirant N structures au hasard, parmi les $(M-m)^3$ structures possibles. Il n'est pas évident de fournir une expression analytique simple pour la fonction $\mathit{nbrdist}(x,y)$ qui donne le nombre moyen d'objets distincts obtenus en faisant x tirages parmi y objets mais une étude expérimentale montre qu'elle vérifie à peu près l'identité [9]

$$\mathit{nbrdist}(k\,x, k\,y) \;=\; k\;\mathit{nbrdist}(x,y). \tag{2.22}$$

2. Nous approximons la valeur $\mathit{nbrdist}(x,y)$ par celle de l'expression

$$\frac{x\,y}{x\;+\;y}.$$

8. Nous supposons que la collection n'utilise qu'un seul symbole de fonction, comme dans notre étude expérimentale. Cela simplifie les calculs mais ceux-ci peuvent être adaptés à d'autres cas de figure. Cela ne change pas fondamentalement les résultats obtenus.

9. Plus précisément, il existe une fonction $\mathit{nbrdist}^*$ telle que $\lim_{k\to\infty} \frac{\mathit{nbrdist}(k\;x, k\;y)}{k} = \mathit{nbrdist}^*(x,y)$; la fonction $\mathit{nbrdist}^*$ vérifie l'identité 2.22 et le rapport entre $\mathit{nbrdist}^*(x,y)$ et $\mathit{nbrdist}(x,y)$ est proche de 1.

Remarquons qu'on a bien :

$$\frac{(kx)\,(ky)}{(kx)\,+\,(ky)} \;=\; k\,\frac{x\,y}{x\,+\,y}.$$

Une étude expérimentale montre que le rapport entre $nbrdist(x,y)$ et $\frac{x\,y}{x+y}$ reste compris entre 1 et $1,265$, si $x \leq y$. Puisque nous nous contentons d'estimations, le choix de cette approximation est raisonnable.

Munis de l'estimation 2.21 de N_m, nous calculons celle de $Tren_m$, en procédant comme pour l'estimation 2.13. Nous obtenons d'abord :

$$Ren_m \;\approx\; 3\,\frac{N\;(M-m)^2}{N\;+\;(M-m)^3} \tag{2.23}$$

Ensuite, nous estimons la somme

$$\sum_{i=0}^{m-1} 3\,\frac{N\;(M-i)^2}{N\;+\;(M-i)^3}, \tag{2.24}$$

par l'intégrale

$$\int_0^m 3\,\frac{N\;(M-x)^2\,dx}{N\;+\;(M-x)^3}. \tag{2.25}$$

Le calcul de cette intégrale fournit l'estimation :

$$Tren_m \;\approx\; N\,\ln\frac{N+M^3}{N+(M-m)^3} \tag{2.26}$$

Une borne supérieure pour l'estimation du nombre total de renommage est donc :

$$N\,\ln\frac{N+M^3}{N+1} \tag{2.27}$$

Il est intéressant de la spécialiser dans les deux cas particuliers où la collection est creuse ($N = M$) ou dense ($N = M^2$). Dans le premier cas, on obtient :

$$N\ln\frac{N+M^3}{N+1} \;=\; M\ln\frac{M+M^3}{M+1} \;\approx\; 2M\ln M \;\approx\; 1,39M\log_2 M. \tag{2.28}$$

Dans le second cas, on trouve :

$$N\ln\frac{N+M^3}{N+1} \;=\; N\ln\frac{M^2+M^3}{M^2+1} \;\approx\; N\ln M \;\approx\; 0,69N\log_2 M. \tag{2.29}$$

2.5.3.2 Paires d'identifiants correspondant à des structures de même clé

Nous cherchons maintenant à estimer le nombre total de renommages $Tren_m$ lorsqu'on exécute des opérations *unify*. Rappelons que l'application d'une opération *unify* à deux identifiants k et l provoque d'abord l'application de l'opération *substitute* à ces identifiants ; ensuite, l'opération *normalize* est exécutée : elle exécute des opérations *substitute* sur des paires d'identifiants correspondant à deux structures de même clé, jusqu'à ce que la collection soit normalisée. Les deux premiers identifiants k et l sont choisis "au hasard", c'est-à-dire en attribuant la même probabilité d'être choisis à tous les identifiants. Par contre, les identifiants choisis par la suite dépendent des structures contenues dans la collection.

Nous cherchons d'abord à découvrir une estimation des nombres N_m, dans le même style que l'estimation 2.21, mais correspondant au nouveau scénario choisi. Une différence fondamentale avec le scénario de la section 2.5.3.1 est que chaque opération *substitute* déclenchée par les opérations *normalize* diminue le nombre de structures de la collection d'au moins une unité. Les nombres N_m doivent donc décroître plus vite (ce qui se voit clairement sur la figure 2.1). Pour "accélérer" la décroissance de l'expression 2.21, nous l'appliquons de manière itérative, en posant, pour tout m tel que $(0 < m \leq M)$:

$$\begin{aligned} N'_0 &= N, \\ N'_m &= \frac{N'_{m-1}\ (M-m)^3}{N'_{m-1}\ +\ (M-m)^3}. \end{aligned}$$

Nous estimons ensuite :

$$N_m \approx N'_m. \tag{2.30}$$

On montre facilement, par récurrence sur m, que l'on a :

$$N'_m = \frac{1}{\frac{1}{N} + \frac{1}{(M-1)^3} + \ldots + \frac{1}{(M-m)^3}}. \tag{2.31}$$

Pour simplifier, de manière approchée, la formule 2.31, nous remplaçons l'expression

$$\frac{1}{(M-1)^3} + \ldots + \frac{1}{(M-m)^3}, \tag{2.32}$$

par le résultat du calcul de l'intégrale

$$\int_0^m \frac{dx}{(M-x)^3}, \tag{2.33}$$

qui vaut :

$$\frac{1}{2}\left(\frac{1}{(M-m)^2} - \frac{1}{M^2}\right). \tag{2.34}$$

En remplaçant l'expression 2.32 par l'expression 2.34, dans l'expression 2.31, et en simplifiant, on obtient l'estimation suivante :

$$N'_m \;\approx\; \frac{2\,N\,M^2\,(M-m)^2}{N\,M^2 \,+\, (2\,M^2 - N)\,(M-m)^2}. \tag{2.35}$$

Nous procédons maintenant comme précédemment pour calculer les estimations de Ren_m et de $Tren_m$. On obtient d'abord :

$$Ren_m \;\approx\; \frac{3\,N'_m}{M-m} \;=\; \frac{6\,N\,M^2\,(M-m)}{N\,M^2 \,+\, (2\,M^2 - N)\,(M-m)^2}. \tag{2.36}$$

En estimant, comme précédemment, la somme des nombres renommages instantanés par une intégrale, en l'occurrence :

$$\int_0^m \frac{6\,N\,M^2\,(M-x)dx}{N\,M^2 \,+\, (2\,M^2 - N)\,(M-x)^2}, \tag{2.37}$$

on trouve :

$$Tren_m \;\approx\; \frac{3\;N\;M^2}{2\;M^2 - N}\;\ln\frac{2\;M^4}{NM^2 + (2M^2 - N)(M-m)^2}. \tag{2.38}$$

On remarque que la valeur de l'expression 2.38 n'est pas définie si $N = 2M^2$. Le calcul correct de l'intégrale 2.37 donne, dans ce cas : $3\,m\,(2\,M - m)$. Ceci correspond à une collection de départ presque hyper dense.

Comme à la section 2.5.3.1, nous allons simplifier l'approximation 2.38 dans les deux cas importants où la collection de départ est creuse ($N = M$) ou dense ($N = M^2$), avec une attention particulière à la valeur extrême $m = M - 1$. Mais nous allons d'abord montrer que l'approximation 2.38 peut se simplifier en

$$Tren_m \;\approx\; \frac{3\;N}{M}\;m, \tag{2.39}$$

pour les petites valeurs de m, exactement comme précédemment. En posant, pour simplifier les notations,

$$P \;=\; \frac{N\;M^2}{2\;M^2 - N},$$

l'approximation 2.38 peut se réécrire :

$$Tren_m \;\approx\; 3\;P\;\ln\left(1 + \frac{m\;(2\;M - m)}{P + (M-m)^2}\right). \tag{2.40}$$

Si m est petit, par rapport à M, on a, ensuite :

$$\ln\left(1+\frac{m\ (2\ M-m)}{P+(M-m)^2}\right)\ \approx\ \frac{m\ (2\ M-m)}{P+(M-m)^2}\ \approx\ \frac{2\ m\ M}{P+M^2}.$$

D'où :

$$\mathit{Tren}_m\ \approx\ \frac{6\ m\ M\ P}{P+M^2}\ =\ \frac{\frac{6\ m\ M^3\ N}{2\ M^2-N}}{\frac{2\ M^4}{2\ M^2-N}}\ =\ \frac{3\ N}{M}\ m. \tag{2.41}$$

Le calcul de la dérivée de la fonction de m définie par l'approximation 2.36 (de Ren_m) montre que cette fonction passe par un maximum lorsque l'on a :

$$m\ =\ M-\sqrt{P}. \tag{2.42}$$

Si la collection est dense, l'égalité ci-dessus, fournit la valeur $m=0$. Il s'en suit que l'approximation de Ren_m est toujours décroissante et que celle de Tren_m reste toujours inférieure à $\frac{3\ N}{M}\ m$. En posant $m=M-1$, on trouve alors que $3\ N$ est une borne supérieure absolue pour Tren_m. Le remplacement de N par M^2 dans l'approximation 2.38 fournit la valeur

$$3\ N\ \ln 2\ \approx\ 2,1\ N, \tag{2.43}$$

qui est un peu meilleure. On a donc le résultat intéressant que, lorsque la collection de départ est dense, le nombre total final de renommages est linéaire dans le nombre de structures : le facteur en $\log M$ a disparu. Lorsque $N<M^2$, on conserve un facteur logarithmique mais les valeurs obtenues pour les approximations de Tren_m sont inférieures à celles obtenues à la section 2.5.3.1 et cela de manière d'autant plus marquée que la valeur N se rapproche de M^2. Pour $N=M$ et $m=M-1$, on obtient seulement l'approximation

$$\frac{3}{2}\ N\ \ln(2\ M), \tag{2.44}$$

qui est à peine inférieure à la borne supérieure 2.10.

2.5.4 Comparaison de l'estimation en moyenne avec l'étude expérimentale

Nous cherchons maintenant à valider les estimations proposées à la section 2.5.3 en les comparant aux résultats expérimentaux de la section 2.5.2.

Nous choisissons les mêmes valeurs pour le nombre d'identifiants et le nombre de structures que dans la section 2.5.2, c'est-à-dire : $M=1.000$

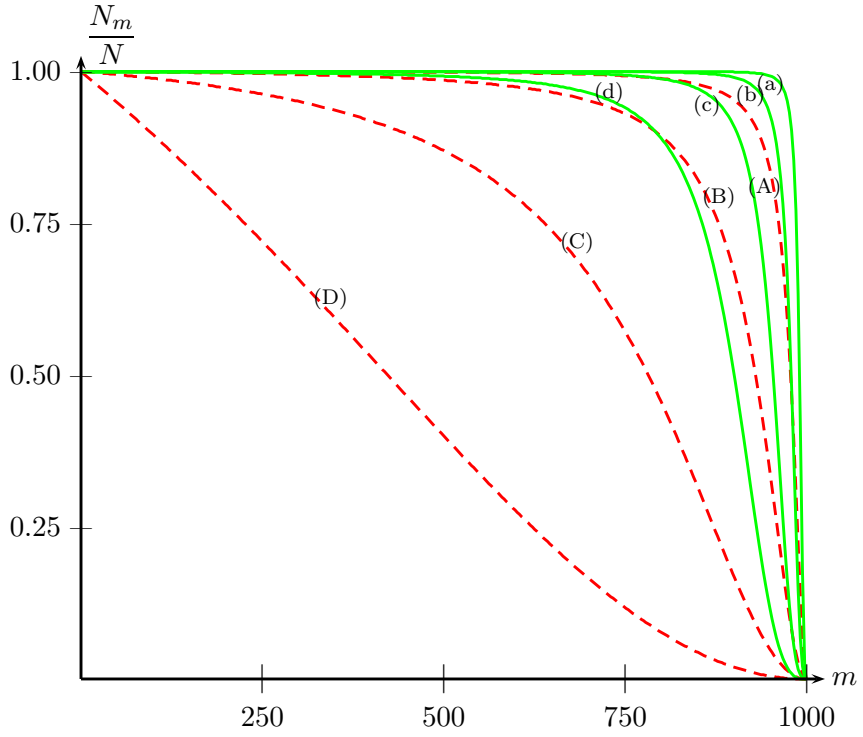


Figure 2.4 – Estimation de N_m, en fonction de m

et $N = 1.000$, 10.000, 100.000, $1.000.000$. Nous calculons les valeurs des estimations proposées à la section 2.5.3 pour ces valeurs et nous les présentons dans trois figures que nous mettons en parallèle avec les figures de la section 2.5.2.

Nous nous intéressons d'abord aux nombres de structures N_m. Les courbes correspondant aux estimations 2.21 et 2.35 de N_m sont représentées à la figure 2.4. Les courbes sont identifiées selon les mêmes conventions qu'à la section 2.5.2.

On constate que les courbes (a), (b), (c), (d) coïncident presque exactement avec les courbes de la section 2.5.2. La courbe (D) est aussi fort proche de celle de l'étude expérimentale. Par contre, les courbes (A), (B), (C) décroissent nettement moins rapidement. Une étude du rapport entre les estimations 2.21 et 2.35 des nombres N_m et leur valeurs expérimentales, montre que, dans le cas des courbes (a), (b), (c), (d), ce rapport reste toujours compris entre 1 et $1,43$. Il reste inférieur à $1,05$ jusqu'à des valeurs de m égales à 964, 925, 847, 715, respectivement, passe par un maximum proche de $1,4$ pour $m = 990$, 980, 957, 910, selon le cas, et redescend ensuite jusqu'à 1. L'estimation 2.21 est donc très précise sauf sur une plage de valeurs limitées de m. Elle fournit en-

core une borne supérieure raisonnable dans cette plage de valeurs. Dans le cas de la courbe (D), le rapport entre l'estimation 2.35 et la valeur expérimentale de N_m reste compris entre $0,88$ et 1 [10]. C'est donc une estimation précise et une borne inférieure. Dans le cas des courbes (A), (B), (C), l'estimation fournit une borne supérieure qui est assez précise pour les valeurs de m qui ne sont pas trop grandes mais le rapport augmente ensuite jusqu'à atteindre des valeurs égales à 9,12 (A), 18,5 (B), 6,06 (C), pour $m = 977$, 952, 945, respectivement. Finalement, le rapport diminue jusqu'à la valeur 1, comme dans les autres cas. L'estimation 2.35 est donc moins satisfaisante que l'estimation 2.21, sauf lorsque la collection de départ est dense.

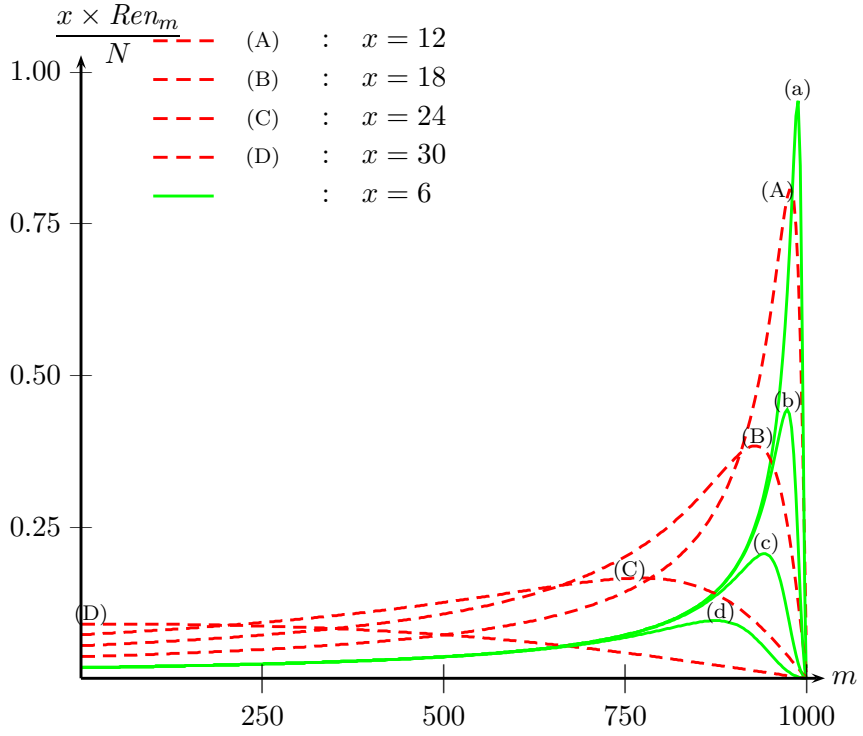


Figure 2.5 – Estimation de Ren_m, en fonction de m

Considérons, à présent, les nombres de renommages "instantanés" Ren_m, représentés à la figure 2.5. À nouveau, les courbes (a), (b), (c), (d) et (D) ont un tracé très semblable à celui de l'étude expérimentale. Les valeurs de m correspondant aux maxima des quatre premières courbes coïncident exactement avec ceux de l'étude expérimentale. L'amplitude des maxima est toutefois à peu près $2,5$ plus grande. Ce facteur ne peut

10. Sauf pour $m = 997$, où il vaut $1,1$.

s'expliquer seulement par le rapport observé entre les valeurs estimées et observées de N_m, discuté, ci-dessus. Pour l'expliquer davantage, nous avons calculé le rapport entre les valeurs de l'expression $\frac{3N_m}{M-m}$ où N_m désigne la moyenne du nombre de structures observées après m opérations *substitute* et les valeurs expérimentales observées pour Ren_m. L'étude de ce rapport nous renseigne sur la pertinence de notre choix d'estimer le nombre Ren_m par la valeur de l'expression $\frac{3N_m}{M-m}$. Pour les courbes (a), (b), (c), (d), on constate que le rapport croît quasi linéairement depuis les valeurs 1,66 (a), 1,12 (b), 1,03 (c), 1,01 (d), jusqu'à des valeurs maximales égales à 2,01 (a), 1,9 (b), $1,8$ (c), 1,66 (d), pour $m = 921$, 911, 841, 731. Le rapport diminue ensuite jusqu'à 1, approximativement. Nous interprétons ces chiffres comme suit : lorsque les identifiants ont la même fréquence d'apparition dans la collection de structures, leur nombre d'occurrences est très proche de $\frac{3N_m}{M-m}$ et notre estimation pour Ren_m est pratiquement exacte. Cette condition est toujours à peu près remplie quand la collection est dense. Lorsque les identifiants ne sont pas répartis uniformément, le fait de renommer l'identifiant qui figure le moins de fois dans la collection implique un nombre moyen de renommages inférieur à $\frac{3N_m}{M-m}$. Par exemple, si l'on suppose une répartition selon une loi de Poisson, on obtient un nombre moyen de renommages égal à $\frac{3N_m}{2(M-m)}$. Vu la manière dont nous avons créé "aléatoirement" les collections de départ, la répartition des identifiants est effectivement proche d'une loi de Poisson quand la collection est creuse ($N = 1.000$) et de plus en plus proche d'une répartition uniforme quand N augmente, ce qui explique très bien les rapports obtenus pour $m = 0$. Lorsque m augmente, la répartition des identifiants est modifiée d'abord de telle sorte que le rapport $\frac{N_m}{M-m}$ diminue mais lorsque m se rapproche de M, la répartition tend vers une distribution uniforme. L'augmentation du rapport se produit tant que N_m reste approximativement égal à N, la diminution a lieu lorsque N se rapproche de $(M-m)^3$.

Considérons, à présent, l'exécution d'opérations *unify* plutôt que celle d'opérations *substitute*, sur des identifiants choisis "au hasard". La comparaison des figures 2.3 et 2.5 montre que notre estimation 2.36 du nombre de renommages est moins satisfaisante que l'estimation proposée pour les opérations *substitute*, sauf, toutefois, lorsque la collection est dense. Les courbes de la figure 2.3 passent par un maximum pour $m =$ 823 (A), 469 (B), 130 (C), 60 (D), alors que celles de la figure 2.5 ont leur maximum pour $m = 977$ (A), 928 (B), 770 (C), 70 (D). L'étude du rapport entre l'estimation $\frac{3N_m}{(M-m)}$ du nombre de renommages, (où N_m est le nombre moyen de structures, mesuré expérimentalement) et le nombre moyen de renommages Ren_m montre que l'estimation $\frac{3N_m}{(M-m)}$

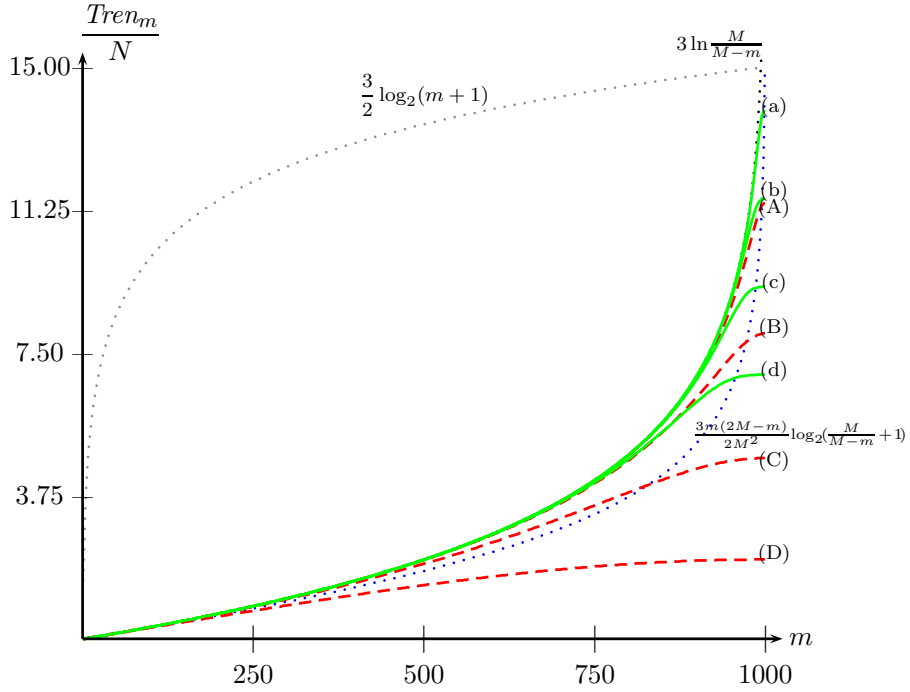


Figure 2.6 – Estimation de $Tren_m$, en fonction de m

est supérieure à la valeur mesurée sauf aux alentours des maximums des courbes de la figure 2.3 (sauf (D)) où elle devient 2 fois inférieure (pour $M = 1.000$, 10.000) et 1,5 fois inférieure (pour $M = 100.000$). L'explication que nous avons trouvée à cette divergence entre la formule estimée et la mesure expérimentale est la suivante. L'exécution d'opérations *unify* au lieu de "simples" opérations *substitute* provoque en priorité le renommage d'identifiants "principaux" figurant dans des structures ayant une clé égale à celle d'une autre structure. Les identifiants les plus nombreux ont donc plus de chances d'être choisis pour un renommage que les autres et ces identifiants sont normalement ceux qui ont déjà participé au plus grand nombre d'opérations *substitute*. Il s'en suit que lorsque la collection de structures commence à contenir un certain nombre de paires de structures de même clé, une sorte d'effet boule de neige se produit : les identifiants les plus nombreux s'affrontent entre eux jusqu'au moment où il ne reste que quelques survivants. Cela correspond aux pics des nombres de renommage observés à la figure 2.3. Cette phase "très active" du traitement se produit d'autant plus tôt que la collection est dense au départ parce que la probabilité d'avoir des structures distinctes de même clé est d'autant plus grande. Si la collection de départ est creuse au sens strict, le phénomène se produit très tard mais

avec d'autant plus de "virulence". Si la collection de départ est dense au sens strict, à l'opposé, le phénomène passe presque inaperçu. La phase "très active" une fois achevée, le reste de l'exécution se déroule avec un nombre moyen de renommages relativement uniforme et inférieur à notre estimation $\frac{3N_m}{(M-m)}$. L'évolution de ce nombre de renommages est étudié en détails à la section 2.5.5.

Plus de détails sur la comparaison entre l'estimation en moyenne et l'étude expérimentale du nombre de renommages sont donnés à l'annexe C. En particulier, on y fournit les courbes montrant les rapports entre les nombres estimés et leur mesure expérimentale.

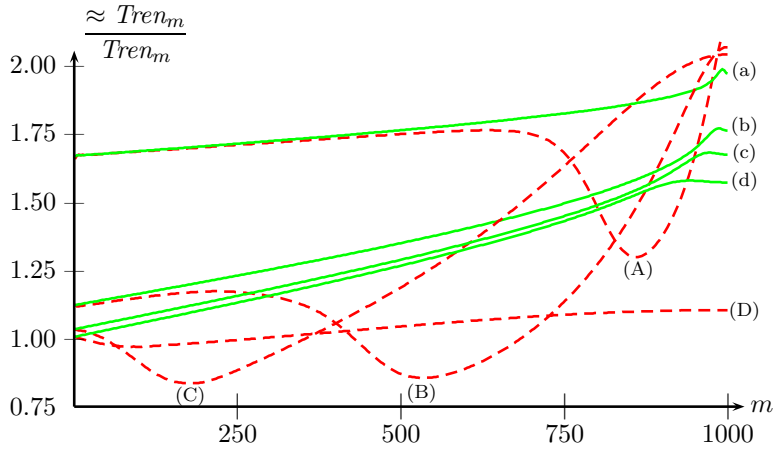


Figure 2.7 – Rapport entre estimation et valeur expérimentale de $Tren_m$

Nous comparons finalement nos estimations 2.13, 2.20, 2.26 et 2.38 des nombres totaux de renommages $Tren_m$ avec les mesures expérimentales de ces nombres, données dans la figure 2.2. Les courbes correspondant à nos estimations sont montrées à la figure 2.6.

Dans le cas d'opérations *substitute* simplement appliquées à des identifiants choisis au hasard, les courbes (a), (b), (c), (d) se comportent sensiblement de la même façon que les courbes correspondant aux mesures expérimentales, elles ne différent de l'approximation la plus simple donnée par l'expression $3 \ln \frac{M}{M-m}$ que pour des valeurs suffisamment grandes de m. On voit aussi que l'approximation plus élaborée 2.20 est meilleure que celle donnée par ces courbes sauf pour les grandes valeurs de m, lorsque la décroissance des nombres N_m devient importante. Dans le cas d'opérations *unify*, les courbes (A), (B), (C), correspondant à l'estimation 2.38 grandissent moins brusquement que les courbes expérimentales mais poursuivent ensuite leur croissance alors que les courbes expérimentales se stabilisent. Pour les collections denses, la courbe

(D) est pratiquement la même selon l'estimation et selon les mesures expérimentales. Pour préciser cette analyse, nous donnons, dans la figure C.4, des courbes décrivant le rapport entre nos estimations et les valeurs observées expérimentalement. Ces courbes montrent que nos estimations sont toujours assez précises, avec un rapport inférieur à 2 et qu'elles fournissent des bornes supérieures aux mesures expérimentales sauf, localement pour $N = 10.000$ et $N = 100.000$. Nos estimations sont particulièrement précises lorsque la collection est dense ($N = 1.000.000$).

2.5.5 Substitution répétée d'un même identifiant

Nous étudions la complexité d'un algorithme qui applique l'opération *substitute* à m paires d'identifiants choisis parmi $m+1 (m \geq 0)$, en choisissant toujours le même identifiant k, comme identifiant "dominant", remplaçant les autres. Un tel algorithme est linéaire dans le nombre de structures contenant certains des m identifiants concernés. En imaginant qu'on sache à l'avance, pour une application donnée, que $m + 1$ identifiants doivent être fusionnés en un seul, on gagne à procéder de la sorte. Ce n'est évidemment pas le cas, en général. Néanmoins, il arrive fréquemment, vu la stratégie habituelle de choix pour k, qu'un seul identifiant apparaisse dans un beaucoup plus grand nombre de structures que les autres, après un nombre suffisant d'opérations *unify* (voir la section 2.5.4). Dès lors, cet identifiant est choisi de plus en plus souvent et la "fin de partie" se déroule à peu près comme étudié ci-dessous.

Si $\tilde{N}_m$ est le nombre total de structures, de la collection de départ, contenant au moins un des m identifiants concernés (autres que k), le nombre de renommages effectués par les m opérations *substitute* est borné par $3\tilde{N}_m$ car chaque structure, ne contenant que 3 identifiants, ne peut être renommée que 3 fois, au plus. $\tilde{N}_m$ étant borné par $3\,N$, on obtient un nombre de renommages égal à $3\,N$ au pire et à $3\,\frac{N}{M}\,m$, en moyenne, si les identifiants ont la même fréquence d'apparition dans les structures.

Nous proposons maintenant de raffiner ces estimations pour nous rapprocher des mesures fournies par les tests expérimentaux. (Les résultats du paragraphe ci-dessus pourront sembler suffisants au lecteur. Il pourra alors passer directement à la section 2.5.6.)

Notons $N_{m\,i}$, le nombre de structures contenant i identifiants différents de k (après $m-1$ opérations *substitute*), avec $i = 0, 1, 2, 3$. À ce stade, une structure contient i identifiants différents de k si elle provient d'un certain nombre de renommages d'une structure contenant i identifiants différents des m identifiants déjà concernés. Notons $\tilde{N}_{m\,i}$, le nombre de

ces structures "originelles". Si l'on suppose que les identifiants ont initialement la même fréquence d'apparition dans les structures, on trouve qu'on a, en moyenne :

$$\tilde{N}_{m\,0} = \frac{Nm^3}{M^3} \qquad (2.45) \qquad \tilde{N}_{m\,1} = \frac{3Nm^2(M-m)}{M^3} \quad (2.46)$$

$$\tilde{N}_{m\,2} = \frac{3Nm(M-m)^2}{M^3} \quad (2.47) \qquad \tilde{N}_{m\,3} = \frac{N(M-m)^3}{M^3} \qquad (2.48)$$

En utilisant la formule, comme précédemment, la formule $\frac{x\,y}{x+y}$ pour estimer le nombre de structures résultant du renommage de x structures dans un ensemble de y structures possibles, on obtient :

$$N_{m\,0} = \frac{Nm^3}{m^3N + M^3} \qquad (2.49) \qquad N_{m\,1} = \frac{3Nm^2(M-m)}{m^2N + M^3} \quad (2.50)$$

$$N_{m\,2} = \frac{3Nm(M-m)^2}{mN + M^3} \quad (2.51) \qquad N_{m\,3} = \frac{N(M-m)^3}{M^3} \qquad (2.52)$$

Justifions, par exemple, la formule 2.51. Le nombre de structures possibles avec 2 occurrences d'un identifiant différent de k, après $m-1$ opérations *substitute*, est $3(M-m)^2$. On obtient donc :[11]

$$\begin{aligned} N_{m\,2} &= \frac{3\tilde{N}_{m\,2}(M-m)^2}{\tilde{N}_{m\,2} + 3(M-m)^2} \\ &= \frac{\frac{9N}{M^3}m(M-m)^4}{\frac{3N}{M^3}m(M-m)^2 + 3(M-m)^2} \\ &= \frac{3Nm(M-m)^2}{mN + M^3} \end{aligned}$$

Au total, nous estimons le nombre de structures de la collection, après $m-1$ opérations *substitute* à :

$$N_m = N_{m\,0} + N_{m\,1} + N_{m\,2} + N_{m\,3}$$

Pour estimer le nombre Ren_mde renommages effectués lors de la $m-$ième exécution de l'opération *substitute*, nous estimons que les identifiants autres que k sont uniformément répartis dans les autres structures. Il y en a $M-m$. Notons $Ren_{m\,i}$ le nombre de renommages des structures contenant i identifiants différents de k. On a d'abord, évidemment,

11. Pour $N_{m\,3}$, on sait que les structures sont inchangées. Donc, $N_{m\,3} = \tilde{N}_{m\,3}$ et on ne recourt pas à une estimation.

$Ren_{m\,0} = 0$. Pour $i = 1, 2, 3$, la proportion de structures contenant l'identifiant remplacé à l'étape m est approximativement égale à : $i/(M-m)$. Le nombre approximatif de renommages est alors fourni par les égalités ci-dessous :

$$Ren_m = Ren_{m\,1} + Ren_{m\,2} + Ren_{m\,3} \tag{2.53}$$

$$Ren_{m\,1} = \frac{3Nm^2}{m^2N + M^3} \tag{2.54}$$

$$Ren_{m\,2} = \frac{6Nm(M-m)}{mN + M^3} \tag{2.55}$$

$$Ren_{m\,3} = \frac{3N(M-m)^2}{M^3} \tag{2.56}$$

Une estimation du nombre total $Tren_m$ de renommages effectués par m exécutions de l'opération *substitute* s'obtient en calculant la somme

$$\sum_{k=1}^{m} Ren_k$$

Comme précédemment, nous pouvons nous affranchir du symbole de sommation en le remplaçant par une intégrale de 0 à m, calculable en termes de fonctions "élémentaires".
Nous obtenons ainsi :

$$Tren_m = Tren_{m\,1} + Tren_{m\,2} + Tren_{m\,3} \tag{2.57}$$

avec

$$\begin{aligned} Tren_{m\,1} &= \int_0^m \frac{3Nx^2dx}{x^2N + M^3} \\ &= 3\left(m - a\ \mathrm{arctg}\left(\frac{m}{a}\right)\right) \quad \text{où} \quad a^2 = \tfrac{M^3}{N} \end{aligned} \tag{2.58}$$

$$\begin{aligned} Tren_{m\,2} &= \int_0^m \frac{6Nx(M-x)dx}{xN + M^3} \\ &= 3\frac{m}{N}(2MN + 2M^3 - mN) - 6\frac{M^4(N+M^2)}{N^2}\ln(\frac{mN+M^3}{M^3}) \\ &= 6\ \frac{M(N+M^2)}{N}\left(m - \frac{M^3}{N}\ln\left(\frac{mN+M^3}{M^3}\right)\right) - 3\ m^2 \end{aligned} \tag{2.59}$$

$$Tren_{m\,3} \;=\; \int_0^m \frac{3N(M-x)^2 dx}{M^3} \;=\; \frac{N\left(M^3-(M-m)^3\right)}{M^3} \qquad (2.60)$$

Les formules obtenues sont "quelque peu complexes". Nous allons les simplifier, en les approximant et en les bornant, dans les cas particuliers où $N = M$ et $N = M^2$, avec une attention particulière pour la valeur limite $m = M - 1$, pour laquelle $Tren_m$ est maximale.

Avant d'analyser en détail les égalités 2.58 à 2.60, nous pouvons déjà, globalement, borner $Tren_m$ supérieurement et inférieurement, d'une part, pour justifier plus rigoureusement que

$$Tren_m \;\leq\; 3\,\frac{mN}{M} \qquad (2.61)$$

et, d'autre part, pour montrer qu'on a :

$$Tren_m \;\approx\; 3\,\frac{mN}{M}, \qquad (2.62)$$

si m est petit par rapport à M.

Pour établir l'inégalité 2.61, nous pouvons d'abord vérifier par un calcul simple que

$$Ren_m \;\leq\; 3\,\frac{N}{M} \qquad (2.63)$$

Dès lors,

$$Tren_m \;=\; \int_0^m Ren_x\,dx \;\leq\; \int_0^m 3\,\frac{N}{M}\,dx \;=\; 3\,\frac{mN}{M} \qquad (2.64)$$

Pour justifier la relation 2.62, nous cherchons une borne inférieure à $Tren_m$ en observant que

$$Ren_m \;\geq\; 3\,\frac{NM^2}{m^2N+M^3} \qquad (2.65)$$

Par conséquent,

$$Tren_m \geq \int_0^m 3\,\frac{NM^2}{x^2N+M^3}\,dx = 3\,\frac{M^2}{a}\,\mathrm{arctg}\left(\frac{m}{a}\right) \text{ où } a^2 = \frac{M^3}{N} \qquad (2.66)$$

Si m est petit par rapport à a, ce qui est le cas si $N \leq M^2$ et si m est petit par rapport à M, la valeur $\mathrm{arctg}\left(\frac{m}{a}\right)$ est approximativement égale à $\frac{m}{a}$. Dès lors,

$$3\,\frac{M^2}{a}\,\mathrm{arctg}\left(\frac{m}{a}\right) \;\approx\; 3\,\frac{mM^2}{a^2} \;=\; 3\,\frac{mN}{M}, \qquad (2.67)$$

ce qui, tenant compte des relations 2.61 et 2.66, établit la relation 2.62.
Étudions, à présent, chacune des équations 2.58 à 2.60.
L'équation 2.58 nous montre, pour commencer, que

$$\mathit{Tren}_{m\,1} \;\leq\; 3\;m \;\leq\; 3\;(M-1) \tag{2.68}$$

Par ailleurs, si $m^2N < M^3$, on obtient une borne supérieure pour $\mathit{Tren}_{m\,1}$, en remplaçant, dans la formule 2.58, la fonction arctg par les deux premiers termes de son développement en série :

$$\mathit{Tren}_{m\,1} \;=\; 3\;(m - a\;\mathrm{arctg}(\frac{m}{a})) \;\leq\; 3\;(m - a\;(\frac{m}{a} - \frac{m^3}{3\;a^3})) \;=\; \frac{m^3N}{M^3} \;<\; N \tag{2.69}$$

En combinant les relations 2.68 et 2.71, nous obtenons la borne supérieure :

$$\mathit{Tren}_{m\,1} \;<\; \min(3\;M,\;N) \tag{2.70}$$

Pour finir, on peut spécialiser la formule 2.58 dans les deux cas particuliers importants où $N = M$ et où $N = M^2$ (et $m = M - 1$). Si $N = M$, on a, d'abord : $a = M$ et $\frac{m}{a} \approx 1$. Dès lors,

$$\mathit{Tren}_{m\,1} \approx 3\;(M - 1 - M\;\mathrm{arctg}(1)) \approx 3\;(M - \frac{\pi}{4}M) \approx 0,644\;M = 0,644\;N \tag{2.71}$$

Nous ne développons pas en détail le cas $N = M^2$ car il n'ajoute pas beaucoup de précision à la formule 2.72. On trouve :

$$\mathit{Tren}_{m\,1} \;\approx\; 3\;(M - \frac{\pi}{2}\sqrt{M}) \;<\; 3\;\sqrt{N} \tag{2.72}$$

Attachons nous maintenant à l'étude, plus difficile, de la formule 2.59. On constate, expérimentalement, que les deux expressions

$$\frac{m}{N}(2MN + 2M^3 - mN) \quad \text{et} \quad \frac{M^4(N+M^2)}{N^2}\ln(\frac{mN+M^3}{M^3})$$

sont quasi égales lorsque mN est petit par rapport à M^3. Par exemple, si $m = 10$ et $M = N = 10.000$, ces deux expressions valent à peu près $6,600.659.637 \times 10^9$ alors que leur différence est environ égale à $0,02$. Le calcul "naïf", en double précision, de la formule 2.59 donne des résultats complètement erronés. Pour obtenir une borne supérieure de $\mathit{Tren}_{m\,2}$, correcte lorsque $mN < M^3$ et précise lorsque mN est petit par rapport à M^3, il est nécessaire de remplacer, dans l'équation 2.59, la fonction $x \mapsto \ln(1+x)$, par les *quatre* premiers termes de son développement en

série. Nous le faisons dans le cas particulier où $N = M$. On obtient :

$$\begin{aligned} \mathit{Tren}_{m\,2} &= 6\ \frac{M(M+M^2)}{M}\ (m - \frac{M^3}{M}\ln(\frac{mM+M^3}{M^3}))\ -\ 3\ m^2 \\ &= 6\ M(1+M)\ (m - M^2\ln(1+\frac{m}{M^2}))\ -\ 3\ m^2 \\ &\leq 6M(1+M)(m - M^2(\frac{m}{M^2} - \frac{m^2}{2M^4} + \frac{m^3}{3M^6} - \frac{m^4}{4M^8})) - 3m^2 \\ &= 3\ \frac{m^2}{M}\ -\ 2\ \frac{m^3}{M^2}\ \Big(1+\frac{1}{M}\Big)\ +\ \frac{3}{2}\ \frac{m^4}{M^4}\ \Big(1+\frac{1}{M}\Big) \end{aligned} \tag{2.73}$$

Nous pouvons maintenant particulariser la formule 2.73 dans le cas où $m = M - 1$, ce qui donne :

$$\begin{aligned} \mathit{Tren}_{m\,2} &\leq 3\frac{(M-1)^2}{M}\ -\ 2\frac{(M-1)^3}{M^2}\Big(1+\frac{1}{M}\Big)\ +\ \frac{3}{2}\frac{(M-1)^4}{M^4}\Big(1+\frac{1}{M}\Big) \\ &= M\ -\ \frac{1}{2}\ -\ \frac{3}{2M}\ -\ \frac{1}{M^2}\ +\ \frac{5}{M^3}\ -\ \frac{9}{2M^4}\ +\ \frac{3}{2M^5} \\ &< M \end{aligned} \tag{2.74}$$

De l'équation 2.76, vraie si $N = M$, on peut déduire qu'on a, plus généralement :

$$\mathit{Tren}_{m\,2}\ <\ N \tag{2.75}$$

En effet, il est facile de voir, en examinant l'intégrale qui définit $\mathit{Tren}_{m\,2}$ (définition 2.58) que le rapport $\frac{\mathit{Tren}_{m\,2}}{N}$ décroît lorsque N augmente. Nous venons d'établir que ce rapport est inférieur à 1 lorsque $N = M$. Il est donc *a fortiori* inférieur à 1 pour de plus grandes valeurs de N, ce qui prouve l'inégalité 2.75.

Le nombre N est une borne supérieure précise pour $\mathit{Tren}_{m\,2}$ seulement si N est proche de M (avec $m = M - 1$). Le rapport $\frac{\mathit{Tren}_{m\,2}}{N}$ diminue ensuite lentement. Nous pouvons l'estimer pour $N = M^2$ en évaluant

directement la formule 2.59, en y remplaçant m et N par leurs valeurs :

$$\begin{aligned} Tren_{m\,2} &= 12M\ (M-1-\frac{M^3}{M^2}\ln(\frac{(M-1)M^2+M^3}{M^3}))\ -\ 3(M-1)^2 \\ &\approx 12\ M\ (M-M\ \ln 2)\ -\ 3\ M^2 \\ &= 3\ M^2\ (3-4\ \ln 2) \\ &\approx 0,682\ M^2 \\ &= 0,682\ N \end{aligned} \tag{2.76}$$

m	N_m	$\approx N_m$	Ren_m	$\approx Ren_m$	$Tren_m$	$\approx Tren_m$	$3\frac{N}{M}m$
5	100	100	2,8	3,0	14	15	15
10	100	100	3,0	3,0	29	30	30
15	100	100	2,8	3,0	43	45	45
20	99	99	2,9	3,0	57	60	60
25	98	98	2,8	3,0	72	75	75
30	96	96	2,8	3,0	86	90	90
35	94	93	2,9	3,0	100	105	105
40	91	90	2,8	2,9	115	119	120
45	87	85	2,7	2,9	129	134	135
50	83	80	2,8	2,8	142	148	150
55	77	73	2,8	2,8	156	162	165
60	71	66	2,6	2,7	170	176	180
65	64	59	2,8	2,6	183	189	195
70	56	50	2,5	2,5	197	202	210
75	48	42	2,5	2,4	209	214	225
80	39	33	2,5	2,2	222	226	240
85	30	24	2,4	2,1	234	237	255
90	20	15	2,3	1,9	245	247	270
95	10	7	2,2	1,7	256	256	285
99	2	1	2,1	1,5	265	262	297

Table 2.1 – Substitution répétée d'un identifiant pour $M = 100$ et $N = 100$

Il nous reste à examiner la formule 2.60 mais celle-ci est suffisamment explicite pour ne pas nécessiter beaucoup de commentaires. On vérifie immédiatement que N est une borne supérieure pour $Tren_{m\,3}$ et qu'elle

est précise lorsque $m = M - 1$.

m	N_m	$\approx N_m$	Ren_m	$\approx Ren_m$	$Tren_m$	$\approx Tren_m$	$3\frac{N}{M}m$
5	998	999	29,2	30,0	148	150	150
10	993	993	29,4	29,9	296	300	300
15	981	978	29,1	29,8	442	449	450
20	957	952	30,0	29,5	589	597	600
25	921	912	29,0	29,0	735	743	750
30	872	860	28,2	28,4	876	887	900
35	813	798	27,3	27,5	1 014	1 027	1050
40	743	729	26,3	26,5	1 146	1 162	1200
45	668	653	25,3	25,3	1 274	1 291	1350
50	588	574	24,4	23,9	1 397	1 414	1500
55	507	494	22,1	22,4	1 511	1 530	1650
60	427	415	20,5	20,7	1 617	1 638	1800
65	348	338	19,0	18,9	1 715	1 737	1950
70	273	265	16,9	17,0	1 805	1 827	2100
75	205	198	15,0	14,9	1 883	1 907	2250
80	143	138	12,7	12,7	1 951	1 976	2400
85	91	87	10,4	10,4	2 007	2 033	2550
90	48	46	7,8	7,9	2 052	2 079	2700
95	18	16	5,2	5,4	2 083	2 113	2850
99	2	1	3,1	3,3	2 099	2 130	2970

Table 2.2 – Substitution répétée d'un identifiant pour $M = 100$ et $N = 1000$

Nous collectons maintenant les résultats obtenus pour estimer le nombre de renommages total $Tren_m$. En général, pour des valeurs quelconques de N ($M \leq N \leq M^2$), on a :

$$Tren_m \ < \ 2\ N + \min(3\ M,\ N) \tag{2.77}$$

Pour $N = M$, on obtient le résultat plus précis (lorsque $m = M - 1$) :

$$Tren_m \ \approx \ 2,644\ M \ = \ 2,644\ N \tag{2.78}$$

et, pour $N = M^2$:

$$Tren_m \ \approx \ 1,682\ N + 3\ \sqrt{N} \tag{2.79}$$

Il paraît donc clair que le rapport du nombre total de renommages sur N décroît lorsque le rapport $\frac{N}{M^3}$ augmente. Nous avons vu aussi que ce

m	N_m	$\approx N_m$	Ren_m	$\approx Ren_m$	$Tren_m$	$\approx Tren_m$	$3\frac{N}{M}m$
5	9 959	9 882	296, 6	298, 5	1 484	1 498	1500
10	9 772	9 552	294, 5	293, 6	2 961	2 979	3000
15	9 389	9 029	289, 7	285, 3	4 421	4 428	4500
20	8 853	8 374	282, 1	274, 4	5 847	5 828	6000
25	8 218	7 637	272, 8	261, 3	7 231	7 168	7500
30	7 519	6 854	261, 8	246, 6	8 564	8 439	9000
35	6 777	6 053	249, 8	230, 6	9 837	9 632	10500
40	6 011	5 258	235, 2	213, 7	11 042	10 744	12000
45	5 239	4 487	220, 5	196, 0	12 175	11 768	13500
50	4 475	3 752	204, 6	177, 9	13 230	12 703	15000
55	3 736	3 066	186, 8	159, 5	14 199	13 546	16500
60	3 034	2 437	168, 6	140, 9	15 080	14 297	18000
65	2 380	1 873	149, 5	122, 4	15 867	14 956	19500
70	1 788	1 378	130, 3	104, 1	16 557	15 522	21000
75	1 265	956	109, 7	86, 0	17 147	15 997	22500
80	822	610	88, 7	68, 3	17 632	16 382	24000
85	467	342	67, 1	51, 1	18 011	16 680	25500
90	207	151	45, 1	34, 4	18 282	16 894	27000
95	50	37	23, 2	18, 3	18 442	17 025	28500
99	2	1	5, 3	6, 0	18 490	17 074	29700

Table 2.3 – Substitution répétée d'un identifiant pour $M = 100$ et $N = 10\,000$

rapport tend vers 1 lorsque N tend vers M^3. Il nous est donc permis de suggérer que la formule

$$3\,\frac{\log M}{\log N}\,N \tag{2.80}$$

fournit une relativement bonne approximation du nombre total de renommages effectués lorsque qu'on effectue $M-1$ opérations *substitute* en choisissant toujours le même identifiant k comme identifiant dominant.

Nous terminons cette section en confrontant nos estimations "théoriques" à des tests "empiriques". Nous avons construit "aléatoirement" des collections de structures utilisant $M = 100$ identifiants et $N = 100, 1.000, 10.000$ structures. Ces collections sont normalisées, au départ. Les collections telles que $N = M = 100$ représentent simplement M termes non équivalents pour la congruence dénotée par la collection de structures. Les collections telles que $N = 1.000, 10.000$ sont obtenues en

ajoutant à une telle collection (pour laquelle $N = 10$), des structures "tirées au hasard" mais ayant des clés distinctes. Pour chacun des trois tests, 1.000 collections ont été construites puis manipulées par un algorithme appliquant $M - 1$ fois l'opération *substitute* au même identifiant et à un autre identifiant choisi "au hasard". Après chaque application de l'opération *substitute*, le nombre de structures de la collection, le nombre de renommages effectués par l'opération et le nombre de renommages effectués depuis le début ont été collectés. Les tableaux 2.1, 2.2 et 2.3 présentent les moyennes de ces mesures pour les 1.000 collections considérées, dans les colonnes N_m, Ren_m et $Tren_m$, respectivement. La colonne m contient le nombre d'opérations *substitute* déjà effectuées. Les colonnes $\approx N_m$, $\approx Ren_m$ et $\approx Tren_m$ sont les estimations des valeurs N_m, Ren_m et $Tren_m$, selon les formules proposées dans cette section. Les valeurs ont été arrondies à l'unité ou à la première décimale, pour le nombre de renommages. Nous laissons au lecteur le soin de constater aussi bien le fait que nos estimations sont proches des mesures expérimentales, que le fait qu'elles concordent avec les valeurs simplifiées que nous en avons données (particulièrement les formules 2.62, 2.63, 2.78 et 2.79). Les écarts constatés entre les mesures expérimentales et leurs estimations sont principalement dues à la façon dont nous estimons le nombre N_m de structures à l'étape m.

2.5.6 Résolution d'équations entre termes

Nous appliquons maintenant les résultats de la sous-section précédente au cas de la résolution d'équations entre termes. Ces résultats s'appliquent plus directement si les termes figurant dans les équations ont été créés préalablement. Supposons donc que l'on résolve un système d'équations utilisant en tout M sous-termes. Les résultats de la sous-section précédente nous indiquent que le nombre de renommages nécessaires sera borné par $\frac{3}{2} M \log M$. Le temps d'exécution total sera la somme du temps nécessaire pour transformer les équations $s = t$ en paires d'identifiants i_s, i_t, par application de l'opération *toSet*, augmenté du temps nécessaire pour appliquer l'opération *unify* à ces paires d'identifiants. Ce temps d'exécution sera donc, au pire, de l'ordre de $O(Teq + M \log M)$, où Teq représente la somme des tailles des équations.

Nous donnons à présent des résultats expérimentaux qui confirment ces estimations théoriques et suggèrent même un temps d'exécution linéaire en pratique, c'est-à-dire en $O(Teq + M)$. Ces résultats montrent aussi qu'on améliore la performance de l'algorithme de résolution des équations si l'on part d'une collection de structures vide et qu'on y ajoute seulement au fur et à mesure, les structures nécessaires à la représenta-

tion des termes utilisés par les équations. On montre enfin que l'ordre dans lequel les équations sont traitées influence l'efficacité de l'algorithme car la résolution d'équations de petite taille donne plus rapidement naissance à une collection de structures dense.

2.5.6.1 Étude expérimentale

Nous avons mené les expérimentations suivantes : nous résolvons un système de 1.000.000 d'équations entre des termes construits à partir de la constante a et du symbole de fonction binaire f. Nous considérons seulement des termes de hauteur inférieure ou égale à 5, où la hauteur d'un terme $f(s,t)$ est égale au maximum de hauteurs de s et de t, plus un, et la hauteur de a est égale à 0. Les nombres de termes de hauteur 0, 1, 2, 3, 4, 5 sont respectivement : 1, 1, 3, 21, 651, 457.653. Il y a donc 458.330 termes, de ce genre, en tout. Les équations sont choisies "aléatoirement" en "tirant au hasard" 1.000.000 de paires de ces termes. Lors de ces tirages, chaque terme reçoit exactement la même probabilité d'être choisi, ce qui est important à noter pour bien comprendre la portée des résultats obtenus, car le fait de choisir plus souvent des termes de petite taille accélèrerait très significativement l'algorithme de résolution d'équations.

Le tableau 2.4 fournit les résultats relatifs à l'exécution de 1.000.000 d'équations, telles que décrites ci-dessus, avec création préalable des termes figurant dans les équations. En réalité, cent groupes d'équations différents ont été utilisés et les chiffres indiqués sont des moyennes sur ces cent exécutions. Les trois premières colonnes du tableau concernent la création d'une collection de structures représentant les termes des équations. La case *# id/str* d'une ligne donne le nombre de sous-termes figurant dans les $(\#\ eq) \times 1.000$ premières équations, et la case t_{cr} donne le temps nécessaire à la création de leur représentation, en secondes. Il y a exactement un identifiant et une structure par sous-terme. On constate que le temps de création est proportionnel au nombre d'équations traitées et que le nombre de termes créés est presque égal au nombre total de termes possibles. Une alternative à la création préalable des termes à partir des équations est de créer une seule fois chaque terme possible mais cela ne change quasi rien aux mesures obtenues pour la création des termes ou par la suite, lors de la résolution des équations.

Viennent alors quatre colonnes fournissant, en fonction du nombre d'équations traitées (*# eq*), les mesures suivantes : le nombre (*# sub*) d'opérations *substitute* exécutées à l'intérieur d'opérations *normalize* (i.e., non dues à la première application de l'opération *substitute*, lors d'une opé-

# eq	# id/str	t_{cr}	# sub	# id	# str	# ren	t_{tot}	t_{eq}	t_{sub}	t_s/r
10	20 218	0.05	13	442 484	452 484	10 033	0.05	0.03	0.02	2.36
20	38 925	0.1	79	432 418	452 418	20 209	0.11	0.06	0.04	2.34
30	56 835	0.14	79	422 418	452 418	30 323	0.17	0.1	0.07	2.32
40	73 980	0.19	118	412 379	452 379	40 617	0.22	0.13	0.09	2.31
50	90 378	0.23	158	402 339	452 339	51 049	0.28	0.16	0.11	2.3
60	106 091	0.28	224	392 273	452 273	61 699	0.34	0.2	0.14	2.29
70	121 122	0.32	963	381 534	451 534	73 967	0.4	0.23	0.16	2.28
80	135 516	0.37	1 002	371 494	451 494	85 005	0.46	0.26	0.19	2.28
90	149 295	0.41	1 106	361 391	451 391	96 459	0.52	0.3	0.21	2.27
100	162 481	0.45	1 265	351 232	451 232	108 358	0.58	0.33	0.24	2.26
110	175 112	0.49	1 449	341 048	451 048	120 688	0.64	0.37	0.27	2.25
120	187 202	0.53	1 567	330 930	450 929	133 325	0.7	0.4	0.29	2.24
130	198 779	0.57	1 684	320 813	450 813	146 472	0.76	0.43	0.32	2.23
140	209 876	0.62	4 297	308 250	446 861	169 059	0.84	0.47	0.37	2.22
150	220 490	0.66	4 482	298 165	446 675	183 491	0.9	0.5	0.4	2.2
160	230 651	0.7	7 671	285 127	441 936	209 883	0.99	0.54	0.45	2.19
170	240 375	0.73	8 004	274 994	441 599	226 447	1.06	0.57	0.48	2.18
180	249 678	0.77	11 359	261 904	436 509	255 635	1.15	0.6	0.54	2.16
190	258 579	0.81	14 734	248 895	431 300	286 947	1.24	0.64	0.6	2.14
200	267 111	0.85	16 832	237 197	429 202	315 182	1.33	0.67	0.65	2.12
210	275 269	0.89	140 043	106 266	200 786	870 573	2.28	0.71	1.57	1.92
220	283 088	0.93	242 692	2 299	4 526	1 330 041	3.06	0.73	2.33	1.75
230	290 575	0.96	244 966	11	32	1 340 225	3.1	0.75	2.35	1.75
240	297 745	1.0	244 968	9	27	1 340 232	3.12	0.77	2.35	1.75
250	304 604	1.04	244 969	7	23	1 340 238	3.14	0.79	2.35	1.75
260	311 175	1.07	244 969	6	21	1 340 241	3.16	0.8	2.35	1.75
270	317 471	1.11	244 972	4	13	1 340 252	3.18	0.82	2.35	1.75
280	323 484	1.14	244 972	3	12	1 340 253	3.2	0.84	2.35	1.75
290	329 244	1.18	244 972	3	10	1 340 255	3.21	0.86	2.35	1.75
300	334 749	1.21	244 972	3	10	1 340 256	3.23	0.88	2.35	1.75
400	378 425	1.55	244 974	1	3	1 340 265	3.42	1.07	2.35	1.75
500	406 665	1.88	244 974	1	2	1 340 266	3.61	1.26	2.35	1.75
1 000	452 497	3.37	244 974	1	2	1 340 267	4.57	2.21	2.35	1.75
`Max`				452497	452497					

Table 2.4 – Résolution d'équations entre termes, créés préalablement

ration *unify*), le nombre # *id* d'identifiants utilisés et le nombre # *str* de structures contenues dans la collection de structures courante, et le nombre de renommages de structures effectués depuis le début (# *ren*). L'analyse de ces quatre colonnes permet de voir que l'algorithme passe par trois phases bien distinctes. Lors de la première phase, c'est-à-dire la résolution des 200.000 premières équations, l'opération de normalisation n'est que peu "productive" : le nombre d'identifiants diminue à peu près du nombre d'équations traitées et le nombre de structures reste quasi constant ; les structures initiales sont simplement réunies deux par deux dans les ensembles de structures E_i, par renommage. La deuxième phase de l'algorithme a lieu entre les équations 200.000 et 220.000. Lors de leur

résolution, la collection de structures est, pour ainsi dire, "chamboulée" de fond en comble. La majorité des opérations *substitute* (environ 230.000) sont effectuées lors de cette phase et elles le sont presque toutes lors des opérations de normalisation. La collection de structures est réduite à 2 299 identifiants et 4 526 structures. Enfin, plus de 1.000.000 de renommages sont effectués : presque tous, à nouveau. La troisième et dernière phase de l'algorithme marque un grand "retour au calme". Sur les 780.000 équations restantes, presque toutes sont devenues "triviales" car leurs termes gauche et droit, étant égaux dans la théorie définie par les équations qui les précèdent, sont traduits, via l'opération *toSet*, par le même identifiant. L'examen plus global de ces chiffres nous montre finalement que le nombre total de renommages est à peu près égal à trois fois le nombre M de structures (et d'identifiants) de la collection de structures de départ. Ce qui suggère une complexité globale en $O(Teq + M)$.

Les quatre dernières colonnes du tableau concernent les temps d'exécution : la colonne t_{tot} fournit, en secondes, le temps d'exécution total pour résoudre $\#\ eq$ équations ; c'est la somme du temps t_{eq} de calcul des identifiants des termes gauche et droit des équations et du temps t_{sub} d'exécution des opérations *unify*. Le quotient entre ce dernier temps et le nombre d'opérations de renommage effectuées est donné par la colonne t_s/r, en micro secondes. Nous constatons que le temps total de résolution des équations (t_{tot}) vaut moins de $1,5$ fois celui de création des termes (t_{cr}), ce qu'on peut réexprimer en disant que "ça ne coûte pas beaucoup plus cher de résoudre les équations que de créer la collection de structures qui représente leurs termes". Lors de la première phase de l'exécution de l'algorithme, les temps t_{eq} et t_{sub} sont quasi égaux. Le temps d'exécution de la troisième phase est entièrement utilisé pour le calcul des identifiants alloués aux termes. Ce n'est que lors de la seconde phase de l'algorithme que le temps de calcul pour la résolution des équations l'emporte, et de beaucoup, sur le calcul des identifiants : environ 30 fois plus. Mais cette phase est transitoire et courte. La colonne t_s/r confirme le fait que le temps t_{sub}, de résolution des équations, est en $O(\#\ ren)$, c'est-à-dire du même ordre que le nombre de renommages. Le fait que les nombres t_s/r décroissent légèrement lors de la deuxième phase de l'exécution s'explique par le fait que les tables de hachage sont vidées de la plupart de leurs éléments, ce qui fait diminuer le temps d'accès à ces tables.

Examinons à présent le tableau 2.5. Celui-ci analyse les résultats obtenus en appliquant aux mêmes équations la méthode de résolution qui est la plus simple et la plus naturelle pour les collections de structures.

# eq	# id/str	t_{cr}	# sub	# id	# str	# ren	t_{tot}	t_{eq}	t_{sub}	t_s/r
10	20 218	0.05	0	10 218	20 218	10 006	0.05	0.05	0.0	0.69
20	38 925	0.1	0	18 924	38 924	20 055	0.11	0.09	0.01	0.67
30	56 835	0.14	0	26 834	56 834	30 168	0.16	0.14	0.02	0.68
40	73 980	0.19	1	33 977	73 977	40 390	0.21	0.18	0.02	0.69
50	90 378	0.23	2	40 372	90 372	50 750	0.26	0.23	0.03	0.7
60	106 091	0.28	5	46 079	106 079	61 284	0.31	0.27	0.04	0.72
70	121 122	0.32	57	51 044	121 044	72 300	0.37	0.31	0.05	0.74
80	135 516	0.37	60	55 417	135 417	83 272	0.42	0.35	0.06	0.76
90	149 295	0.41	71	59 163	149 162	94 563	0.47	0.4	0.07	0.78
100	162 481	0.45	90	62 304	162 304	106 215	0.52	0.44	0.08	0.8
110	175 112	0.49	117	64 881	174 881	118 270	0.58	0.48	0.09	0.82
120	187 202	0.53	135	66 918	186 918	130 735	0.63	0.52	0.11	0.85
130	198 779	0.57	158	68 436	198 435	143 721	0.69	0.56	0.12	0.88
140	209 876	0.62	721	68 932	207 543	161 656	0.75	0.6	0.14	0.91
150	220 490	0.66	763	69 472	217 982	175 846	0.81	0.64	0.16	0.93
160	230 651	0.7	1 520	68 843	225 652	196 729	0.88	0.68	0.19	0.97
170	240 375	0.73	1 611	68 443	235 048	212 889	0.94	0.72	0.21	1.0
180	249 678	0.77	2 479	66 849	241 454	236 832	1.01	0.76	0.25	1.04
190	258 579	0.81	3 393	64 816	247 221	262 855	1.09	0.8	0.29	1.07
200	267 111	0.85	4 127	62 584	254 589	288 835	1.17	0.84	0.33	1.12
210	275 269	0.89	36 912	28 289	122 808	627 968	1.74	0.87	0.86	1.31
220	283 088	0.93	64 288	620	2 847	905 598	2.2	0.89	1.3	1.44
230	290 575	0.96	64 901	2	24	911 997	2.23	0.91	1.31	1.44
240	297 745	1.0	64 901	2	20	912 001	2.25	0.93	1.31	1.44
250	304 604	1.04	64 902	2	18	912 004	2.27	0.95	1.31	1.44
260	311 175	1.07	64 902	2	17	912 005	2.29	0.97	1.31	1.44
270	317 471	1.11	64 902	1	11	912 012	2.31	0.99	1.31	1.44
280	323 484	1.14	64 902	1	10	912 013	2.33	1.01	1.31	1.44
290	329 244	1.18	64 902	1	9	912 014	2.35	1.03	1.31	1.44
300	334 749	1.21	64 902	1	8	912 015	2.37	1.05	1.31	1.44
400	378 425	1.55	64 902	1	3	912 022	2.56	1.24	1.31	1.44
500	406 665	1.88	64 902	1	2	912 023	2.75	1.43	1.31	1.44
1 000	452 497	3.37	64 902	1	2	912 023	3.7	2.38	1.31	1.44
Max				70182	269556					

Table 2.5 – Résolution d'équations entre termes, non préalablement créés

Cette fois, on part d'une collection de structures vide et on crée la représentation de termes "à la demande" en même temps que l'on résout les équations. On constate immédiatement que les résultats obtenus de cette façon sont significativement meilleurs, tant en utilisation de la mémoire que du point de vue du temps d'exécution. Le temps utilisé pour la résolution des équations est diminué de 44%. Le nombre maximum de structures utilisées l'est de 40% et celui des identifiants de 85%. Le nombre de renommages est réduit de 32%. Cette fois, le temps complet de traitement des équations n'est qu'à peine supérieur au temps de création d'une représentation des termes. On peut enfin remarquer que le rapport

t_s/r est inférieur à celui obtenu précédemment et qu'il croît progressivement. À nouveau nous l'expliquons par le remplissage des tables de hachage qui est d'abord nul et qui augmente progressivement, jusqu'au début de la seconde phase de l'algorithme.

# *eq*	# *id/str*	t_{cr}	# *sub*	# *id*	# *str*	# *ren*	t_{tot}	t_{eq}	t_{sub}	t_s/r
10	17 409	0.04	1	7 372	17 372	10 116	0.05	0.04	0.0	0.68
20	36 263	0.09	6	16 104	36 104	20 230	0.1	0.09	0.01	0.68
30	54 312	0.14	18	23 945	53 945	30 472	0.16	0.14	0.02	0.7
40	71 591	0.18	53	30 918	70 918	40 988	0.21	0.18	0.02	0.71
50	88 119	0.23	111	37 038	87 038	51 794	0.26	0.23	0.03	0.73
60	103 954	0.27	206	42 351	102 351	62 993	0.32	0.27	0.04	0.76
70	119 098	0.32	1 435	45 722	113 622	83 187	0.39	0.31	0.07	0.8
80	133 601	0.36	2 599	48 383	124 383	102 522	0.45	0.36	0.09	0.84
90	147 483	0.41	3 813	50 277	133 977	122 110	0.52	0.4	0.12	0.88
100	160 765	0.45	5 377	51 128	142 128	143 871	0.59	0.44	0.14	0.93
110	173 489	0.49	10 465	47 755	139 055	194 393	0.7	0.48	0.22	1.02
120	185 667	0.53	21 200	38 042	116 042	293 431	0.89	0.51	0.38	1.14
130	197 327	0.57	43 558	16 066	51 167	490 930	1.24	0.54	0.69	1.33
140	208 504	0.61	54 243	5 441	18 042	587 915	1.41	0.57	0.84	1.41
150	219 193	0.66	59 682	1	2	637 486	1.51	0.59	0.92	1.44
500	406 500	1.88	59 682	1	2	637 486	2.2	1.28	0.92	1.44
1 000	452 497	3.38	59 682	1	2	637 486	3.19	2.27	0.92	1.44
`Max`				57032	175833					

Table 2.6 – Résolution d'équations, triées sur la hauteur des termes

Nous allons montrer maintenant que le comportement de l'algorithme change significativement si on change l'ordre dans lequel les équations sont traitées. Le tableau 2.6 donne des résultats relatifs au traitement des mêmes équations, sans création préalable des termes. Ce qui change par rapport aux exécutions analysées dans le tableau 2.5, c'est que l'on a trié les équations de façon à placer au début celles qui contiennent les termes les plus petits. Plus précisément, on met en tête celles dont le terme gauche ou le terme droit a une hauteur égale à 0, puis celles pour lesquelles un de ces termes a une hauteur égale à 1, et ainsi de suite. Les équations dont les deux termes ont une hauteur maximale, égale à 5, sont placées à la fin. (Elles sont, de loin, les plus nombreuses.) La comparaison des tableaux 2.6 et 2.5 montre encore une amélioration significative des performances. Le nombre de renommages et le temps de résolution des équations sont encore diminués de 30%. À présent, le temps total t_{tot} d'exécution de l'algorithme est même inférieur au temps t_{cr} de création des termes.

Il est aussi intéressant de montrer, *a contrario*, ce qui se passe si on trie les équations dans l'ordre inverse, en commençant par les équations avec les plus grands termes. Les résultats de cette variante sont donnés

# eq	# id/str	t_{cr}	# sub	# id	# str	# ren	t_{tot}	t_{eq}	t_{sub}	t_s/r
100	162 704	0.47	0	62 705	162 704	105 577	0.54	0.45	0.08	0.81
200	267 376	0.88	0	67 377	267 376	256 819	1.14	0.86	0.28	1.09
300	334 980	1.25	0	40 289	334 980	650 365	2.14	1.25	0.88	1.36
400	378 677	1.6	0	19 647	378 677	802 439	2.73	1.62	1.11	1.39
500	406 880	1.93	0	9 411	406 880	867 021	3.17	1.96	1.2	1.39
600	425 106	2.25	0	4 667	425 106	899 171	3.54	2.29	1.24	1.38
700	436 871	2.56	0	2 496	436 871	916 639	3.88	2.61	1.26	1.38
800	444 456	2.86	0	1 506	444 456	926 646	4.21	2.93	1.28	1.38
900	449 360	3.16	0	1 052	449 360	932 607	4.53	3.24	1.28	1.37
910	449 742	3.19	0	1 024	449 742	933 054	4.56	3.27	1.28	1.37
920	450 111	3.22	0	997	450 111	933 483	4.59	3.3	1.28	1.37
930	450 463	3.25	0	973	450 463	933 889	4.62	3.33	1.28	1.37
940	450 799	3.28	0	950	450 799	934 278	4.65	3.36	1.28	1.37
950	451 122	3.31	0	929	451 122	934 649	4.68	3.39	1.28	1.37
960	451 429	3.33	0	909	451 429	934 999	4.71	3.43	1.28	1.37
970	451 724	3.36	0	891	451 724	935 336	4.75	3.46	1.28	1.37
980	452 007	3.39	0	875	452 007	935 655	4.78	3.49	1.28	1.37
990	452 278	3.42	0	860	452 278	935 959	4.81	3.52	1.29	1.37
1 000	452 497	3.45	183	1	2	1 391 019	5.4	3.55	1.84	1.32
Max				70937	452460					

Table 2.7 – Résolution d'équations, triées en ordre décroissant

dans le tableau 2.7. On peut dire que les résultats sont globalement deux fois moins bon que lorsque les équations sont triées en ordre croissant de la hauteur des termes : le nombre de renommages et le temps de résolution des équations sont doublés. On constate aussi que le nombre de structures *# str* reste égal au nombre *# id/str* de sous-termes figurant dans les équations, jusqu'au-delà de l'équation 990.000, et que l'opération de normalisation ne produit aucun effet avant la toute fin de l'exécution : comme la majorité des équations contiennent des termes de hauteur maximale, leur "unification" au sein d'un même ensemble de structures E_i ne peut se propager dans des termes de plus grande taille. Ce n'est qu'à la fin qu'une propagation se produit lors du traitement des dernières équations, qui mettent en relation de petits termes.

2.5.6.2 Remarques générales et conclusion

Nous avons montré expérimentalement, sur une catégorie particulière d'équations entre termes, que l'utilisation des collections de structures pour résoudre ces équations [12] nécessite un nombre d'opérations de renommage qui n'excède jamais trois fois le nombre M de sous-termes figurant dans les équations. Nous avons montré également que le temps

12. C'est-à-dire pour calculer, sous forme de collection de structures, une représentation de la théorie définie par ces équations.

nécessaire à la résolution des équations (exécution des opérations *unify*) est proportionnel au nombre de renommages. Le tout impliquant un temps d'exécution de l'ordre de $O(Teq + M)$ où Teq est la taille totale des équations. Nous pensons que les résultats obtenus sont valables plus généralement car la forme des équations est délibérément défavorable à notre méthode : en augmentant la fréquence des termes de petite taille ou en choisissant certaines autres formes particulières d'équations exprimant des propriétés comme l'associativité ou la commutativité de l'opération f, l'efficacité de la méthode aurait encore été bien meilleure.

Les expériences que nous avons menées montrent également que l'espace mémoire nécessaire pour effectuer la résolution peut être rendu significativement inférieur à l'espace nécessaire pour représenter la totalité des sous-termes des équations, particulièrement si les équations sont résolues en choisissant les plus courtes d'abord. Ceci est particulièrement important pour les applications dans lesquelles on a un nombre très grand, voire potentiellement infini, d'équations à considérer. Il importe alors de trier ou d'engendrer ces équations en ordre croissant de taille, pour créer en un nombre fini d'étapes une collection de structures contenant la totalité des sous-termes des équations. À ce moment-là, la collection de structures ne pourra plus qu'être réduite et on pourra souvent arrêter l'exécution de l'algorithme sans résoudre les équations restantes. Nous utilisons cette approche à la section 3.2, pour construire une représentation de la théorie des équations booléennes portant sur un nombre fini de constantes propositionnelles, en engendrant les équations par "instanciation" d'une axiomatisation du calcul booléen. En réalité, nous n'instancions pas les axiomes avec des termes mais avec des identifiants d'ensembles de structures. Le critère d'arrêt utilisé est qu'aucune instanciation possible des axiomes ne produise plus de nouvelle structure.

2.6 Travaux apparentés

Le travail présenté dans ce chapitre a été essentiellement réalisé “en partant d'une page blanche”, en nous appuyant seulement sur notre connaissance des notions “de base” de l'informatique dont on trouve des exposés classiques, par exemple, dans [Knu97, AU92, LRSC01]. L'intuition sous-jacente à nos collections de structures est de représenter des ensembles de termes égaux (dans certaines interprétations) par des identifiants uniques et d'utiliser ces identifiants pour calculer ou “raisonner” sur ces ensembles de termes, plutôt que sur des objets plus compliqués ou arbitraires tels que des représentants de classes d'équivalence. Notre objectif initial (et final) est de proposer une méthode générale de simplification des expressions consistant à calculer toutes les expressions égales à une expression donnée, dans un cadre logique ou mathématique précis, puis d'en extraire les plus simples selon divers critères tels que, par exemple, la taille de l'expression.

Les travaux apparentés évoqués plus bas, dans cette section, n'ont donc pas été directement à l'origine de notre travail. Leur étude est postérieure à l'élaboration des résultats présentés dans les sections précédentes de ce chapitre. Plusieurs de ces travaux apparentés nous ont été signalés par des collègues ou des lecteurs de versions préliminaires et partielles de ce chapitre [LA16] et du chapitre suivant [LA15].

2.6.1 Calcul de la fermeture congruente d'une relation

Dans cette section, nous mettons notre travail en relation avec les algorithmes de fermeture congruente proposés dans la littérature [Sho78, Kap97, NO80, DST80, NO05]. Nous commençons par une comparaison approfondie avec la méthode de Shostak [Sho78, Kap97] et certaines de ses améliorations (voir, par exemple, [NO05]), parce que les notions utilisées dans ces travaux peuvent être plus directement reliées à notre propre travail que d'autres algorithmes tels que ceux proposés dans [NO80, DST80]. Nous ajoutons une brève comparaison avec ces autres travaux, à la fin de cette section.

2.6.1.1 Présentation simplifiée de la méthode de Shostak

Soient T, un ensemble de termes clos et ρ, une relation entre termes telle que $\rho \subseteq T \times T$. Il n'est pas supposé que $T = dom(\rho)$. (Voir la définition 5. Plus généralement, nous utilisons les définitions et notations de la section 2.2.) La *fermeture congruente* de la relation ρ par rapport à l'ensemble de termes T est la plus petite relation d'équivalence $\equiv$, définie

sur T et contenant ρ, telle que l'on ait :

$$\left.\begin{array}{c} s_i \equiv t_i \ (i=1,2) \\ f(s_1,s_2) \in T \\ f(t_1,t_2) \in T \end{array}\right\} \Longrightarrow f(s_1,s_2) \equiv f(t_1,t_2)$$

pour tous les termes s_i et t_i $(i = 1, 2)$, appartenant à T. On suppose aussi que l'ensemble T est complet pour les sous-termes (voir la définition 4). Le but d'un algorithme de fermeture congruente est de calculer la fermeture d'une relation ρ, par rapport à un ensemble de termes T. La relation ρ peut être vue comme un ensemble fini d'équations entre termes clos. Par convention de langage, nous appelons ci-dessous, *congruence partielle* toute relation pouvant être définie comme fermeture d'une relation ρ par rapport à un ensemble de termes T, comme précisé ci-dessus.

L'idée de la méthode de Shostak est de calculer la fermeture congruente d'une relation à l'aide d'une structure de données du type Union-Find [GF64]. À un moment donné, on a déjà calculé une congruence partielle $\equiv$, à partir d'un certain nombre d'équations et on considère une nouvelle équation $s = t$. On fusionne les classes d'équivalences des termes s et t. Cela augmente le nombre de termes s_i et t_i $(i = 1, 2)$ tels que $s_i \equiv t_i$. On "propage" alors la relation d'équivalence aux paires de termes $f(s_1, s_2)$ et $f(t_1, t_2)$, appartenant à T. Cela rajoute encore de nouveaux couples à la relation d'équivalence. On continue donc à "propager" la relation jusqu'à ce qu'aucun couple ne puisse lui être ajouté.

Pour réaliser systématiquement la "propagation" expliquée ci-dessus, on s'appuie sur la structure de donnée Union-Find qui représente la relation d'équivalence courante. Cette structure fournit pour chaque classe d'équivalence, un élément de référence, appelé *représentant*. Si $t \in T$, nous notons $\overline{t}$, le représentant de la classe d'équivalence de t. À chaque terme $f(t_1, t_2) \in T$, nous faisons correspondre le terme $f(\overline{t}_1, \overline{t}_2)$, appelé la *signature* de $f(t_1, t_2)$. On utilise cette signature pour "détecter" les nouveaux termes équivalents : deux termes deviennent équivalents si leurs signatures sont rendues égales par une modification de la structure Union-Find. Il suffit donc, à chaque modification de la structure Union-Find, de retrouver les signatures qui ont changé et de rendre égaux les termes correspondant à ces signatures.

Nous allons maintenant donner deux algorithmes qui implémentent précisément les principes ci-dessus. Nous donnons d'abord chaque fois, une description de "très haut" niveau des algorithmes puis nous précisons les structures de données concrètement utilisées. Dans ce qui suit, nous notons $sig(t)$, la signature (courante) d'un terme $t \in T$. On notera que

la signature d'un terme de T n'appartient pas nécessairement à T.

2.6.1.2 Premier algorithme

L'algorithme que nous présentons maintenant, tout comme le suivant à la section 2.6.1.3, "traite une seule équation" $s = t$ (ou un seul couple de la relation ρ). Comme précondition, il suppose que la structure Union-Find représente une congruence partielle sur l'ensemble T (fixé) et que $s, t \in T$. L'algorithme étend cette congruence partielle "le moins possible" mais de telle sorte qu'elle contienne la couple $\langle s, t \rangle$. L'algorithme reçoit aussi, pour chaque représentant $\overline{t}$ d'une classe d'équivalence, la liste $\mathit{liste}(\overline{t})$ de tous les termes de T dont la signature contient $\overline{t}$, comme sous-terme direct (voir définition 1).

L'algorithme pour traiter un ensemble d'équations (tous les couples de ρ) crée une structure Union-Find représentant la congruence partielle minimale dans laquelle les termes de T ne sont équivalents qu'à eux-mêmes. Pour tous les termes $t \in T$, il initialise les listes $\mathit{liste}(t)$ à la liste de tous les termes qui contiennent t comme sous-terme direct. Puis il applique l'algorithme ci-dessous, successivement, à chaque équation.

L'algorithme de haut niveau utilise une variable P contenant un ensemble de couples de termes de T. Sa correction est basée sur l'invariant suivant :

$$\forall u, v \in T : (\mathit{sig}(u) = \mathit{sig}(v) \text{ et } \overline{u} \neq \overline{v}) \implies \langle u, v \rangle \in P$$

Outre cet invariant, l'algorithme maintient les propriétés caractéristiques des listes $\mathit{liste}(\overline{t})$. Voici donc l'algorithme :

1. On exécute : $P := \{\langle s, t \rangle\}$.
2. Tant que $P \neq \{\}$, on exécute ce qui suit :
   (a) On prélève une paire $\langle u, v \rangle$, dans P.
   (b) Si $\overline{u} = \overline{v}$, on itère sans exécuter les étapes suivantes.
   (c) On renomme les listes $\mathit{liste}(\overline{u})$ et $\mathit{liste}(\overline{v})$, respectivement, en liste_1 et liste_2.
   (d) On fusionne les classes d'équivalence de u et v.
   (e) On ajoute à P toutes les paires de termes $\langle q, r \rangle$ telles que
      - q appartient à liste_1,
      - r appartient à liste_2,
      - $\mathit{sig}(q) = \mathit{sig}(r)$.
   (f) On fait : $\mathit{liste}(\overline{u}) := \mathit{liste}_1 \cup \mathit{liste}_2$.

La correction de cet algorithme de haut niveau se justifie brièvement comme suit.

1. L'algorithme se termine car le nombre de classes d'équivalence décroît d'une unité à chaque itération, si $\overline{u} \neq \overline{v}$, et le nombre d'éléments de P décroît d'une unité, sinon.
2. L'invariant est respecté initialement car la relation d'équivalence initiale est une congruence et il est maintenu à chaque itération car si la signature d'un terme change et devient égale à celle d'un autre terme, une paire contenant les deux termes est placée dans P [13]. La propriété caractéristique des listes $liste(\overline{t})$ est aussi maintenue puisque l'ensemble des termes dont la signature contenait $\overline{u}$ ou $\overline{v}$ (comme sous-terme direct), avant itération, est égal à l'ensemble des termes dont la signature contient $\overline{u}$ $(= \overline{v})$, après itération.
3. Lorsque l'algorithme se termine, la condition $P = \{\}$, combinée au respect de l'invariant, assure qu'on a :

$$\forall u, v \in T : sig(u) = sig(v) \implies \overline{u} = \overline{v}$$

et il est clair que cette dernière condition implique que la relation d'équivalence courante est une congruence partielle.

Nous avons implémenté cet algorithme est utilisant les mêmes structures de "bas niveaux" que pour notre implémentation des collections de structures (voir section 2.4.4). Chaque terme $f(t_1, t_2)$ de l'ensemble T est représenté par l'identifiant i d'une structure $f(i_1, i_2) : i$ où i_1 et i_2 sont les identifiants représentant les termes t_1 et t_2. Donc, l'ensemble de termes T est globalement représenté par une collection de structures unique. Cette collection n'est pas modifiée lors de l'exécution de l'algorithme ci-dessus. L'algorithme "concret" manipule évidemment les identifiants d'ensembles de structures, non les "termes". L'utilisation de ces identifiants (en fait des entiers) rend immédiate la réalisation d'une structure de données Union-Find pour représenter la relation d'équivalence courante. Elle permet aussi de réaliser efficacement les listes $liste(\overline{u})$ exigées par l'algorithme. Nous utilisons pour cela deux "instances" de la classe `MultiList` (voir section 2.4.4). La première instance s'occupe de tous les termes dont la signature contient $\overline{u}$, comme premier argument. La seconde s'occupe de ceux dont la signature contient $\overline{u}$, comme second argument. Rappelons qu'une seule instance suffit à gérer l'entièreté des listes associées à la totalité des identifiants utilisés par la collection de structures. Ainsi l'opération "abstraite" $liste(\overline{u}) := liste_1 \cup liste_2$ est réalisée en déplaçant chaque identifiant d'un terme figurant dans la liste

13. On considère les paires comme "non ordonnées".

correspondant à celui des termes $\overline{u}$ et $\overline{v}$ qui n'est pas choisi comme nouveau représentant, vers la liste correspondant à l'autre.

Il est démontré dans [Kap97] que la complexité de l'algorithme ci-dessus est en $O(N^2)$ où N est le nombre de termes de T. On peut le comprendre intuitivement en observant que le point 2e de l'algorithme est réalisé concrètement par une double boucle sur les listes $liste_1$ et $liste_2$ et que les termes de T sont progressivement accumulés dans les listes $liste(\overline{u})$ au fur et à mesure de la fusion des classes d'équivalence. La complexité spatiale de l'algorithme est également en $O(N^2)$, ce qui rend l'algorithme inapplicable pour un grand nombre de termes, comme nous allons le montrer à la section 2.6.1.4. Il est possible d'améliorer l'algorithme tel que nous l'avons présenté ci-dessus en remarquant deux choses. D'une part, il n'est pas nécessaire de placer dans l'ensemble P toutes les paires (non ordonnées) $\langle q, r\rangle$ telles que $sig(q) = sig(r)$. Il suffit que, pour chacune d'entre elles, il existe une suite de paires $\langle q_1, r_1\rangle$, $\ldots \langle q_k, r_k\rangle$, avec $q = q_1$, $r_i = q_{i+1}$ $(1 \leq i < k)$, et $r_k = r$. D'autre part, il est possible de maintenir cette condition plus faible, en ne conservant dans les listes $liste(\overline{u})$ qu'un seul terme parmi tous ceux qui ont la même signature. Nous laissons au lecteur le soin de compléter les détails mais nous donnons une évaluation expérimentale de cet algorithme amélioré à la section 2.6.1.4.

2.6.1.3 Second algorithme

Même si son coût peut être substantiellement diminué grâce aux améliorations que nous avons indiquées, la cause principale d'inefficacité de l'algorithme de la section précédente est l'utilisation d'une double boucle. Il est possible de la remplacer par une simple boucle, sur l'une des deux listes $liste_1$ ou $liste_2$ en gardant en mémoire les signatures des termes, en plus des termes eux-mêmes. Supposons que le terme $\overline{v}$ soit choisi comme nouveau représentant après la fusion des classes d'équivalence des termes u et v. Il suffit de parcourir la liste $liste_1$ (ancienne valeur de $liste(\overline{u})$) et de vérifier si la nouvelle signature de chaque terme existe déjà ou pas. Si elle existe, les termes doivent être ajoutés à l'ensemble P. On peut, de plus, maintenir, dans la structure Union-Find, l'équivalence entre chaque terme et sa signature, ce qui permet de remplacer les termes par leur signature dans les listes $liste(\overline{u})$, diminuant la taille de ces listes.

Le nouvel algorithme est construit et justifié sur base de l'invariant suivant :

$$\forall u \in T : \overline{u} \neq \overline{sig(u)} \implies \exists \langle u', v'\rangle \in P : (\overline{u} = \overline{u'} \text{ et } \overline{v'} = \overline{sig(u)}).$$

L'algorithme utilise, cette fois, un ensemble variable de termes que nous

notons $T^\sharp$ et la structure Union-Find maintient une relation d'équivalence sur cet ensemble. La précondition de l'algorithme est que $T^\sharp$ est égal à la réunion de T et des signatures des termes de T. On doit avoir aussi :

$$\forall u \in T : \overline{u} \;=\; \overline{sig(u)}.$$

L'algorithme s'énonce comme suit :

1. On exécute : $P := \{\langle s, t\rangle\}$.
2. Tant que $P \neq \{\}$, on réalise les actions ci-dessous :
   (a) On prélève une paire $\langle u, v\rangle$, dans P.
   (b) Si $\overline{u} = \overline{v}$, on itère sans exécuter les étapes suivantes.
   (c) On fusionne les classes d'équivalence de u et v, en choisissant $\overline{v}$ comme représentant de la classe fusionnée. Soit *liste*, la valeur de $liste(\overline{u})$, avant la fusion.
   (d) Pour chaque terme q de la liste *liste*, on fait ceci : si $sig(q) \in T^\sharp$, on ajoute à P la paire $\langle q, sig(q)\rangle$; sinon, on ajoute $sig(q)$ à $T^\sharp$ et on place $sig(q)$ dans la classe d'équivalence de q et dans la liste $liste(\overline{v})$.
   (e) On retire de $T^\sharp$ tous les termes de *liste* qui n'appartiennent pas à T.

La correction de cet algorithme se montre dans le même style que celle du premier algorithme. Nous l'avons implémenté avec les mêmes structures de données. Nous utilisons la collection de structures pour représenter l'ensemble de termes $T^\sharp$. Plus de termes sont donc représentés. De même, la structure Union-Find doit être étendue pour inclure les identifiants des signatures. Le pas 2e de l'algorithme permet de réduire la taille de la collection de structures et de la structure Union-Find. Il n'est pas nécessaire pour la correction de l'algorithme, "en principe".

Un algorithme très semblable à ce deuxième algorithme est décrit dans [NO05]. Les auteurs de cet article argumentent que sa complexité est en $O(N \log N)$, ce qui est cohérent avec notre étude de complexité pour notre algorithme de résolution d'un ensemble d'équations entre termes (voir la section 2.5.6). Nous montrons à la section 2.6.1.4, ci-dessous, que cette version de l'algorithme de Shostak calcule la fermeture congruente d'un ensemble d'équations d'une manière très semblable à celle de notre algorithme de la section 2.3.3. Néanmoins, notre algorithme utilise moins de mémoire et est conceptuellement plus simple. Il fournit aussi un résultat plus général, c'est-à-dire un modèle de l'ensemble des équations (voir la section 2.5.6 et, en particulier, le point 3 de la section 2.3.3.1).

2.6.1.4 Comparaison expérimentale

Dans cette section, nous comparons expérimentalement les algorithmes des deux sections précédentes et notre algorithme de résolution d'équations entre termes. Nous utilisons comme données de test l'ensemble de termes T décrit à la section 2.5.6.1, soient les 458.330 termes de hauteur inférieure ou égale à 5 que l'on peut construire sur base de la constante a et du symbole de fonction binaire f. Nous générons "aléatoirement" 250.000 équations entre ces termes que nous résolvons (avec notre algorithme) ou dont nous calculons la fermeture congruente (avec différentes variantes de l'algorithme de Shostak).

Le tableau 2.8 compare les temps d'exécution. Chaque ligne de la table fournit les temps nécessaires à chaque algorithme pour résoudre les équations dont le nombre figure dans la colonne $\sharp\ eq$. Les temps sont donnés en secondes. Les colonnes t_{cs}, t_{ol}, $t_{ol'}$, t_{sh}, et $t_{sh'}$ donnent les temps, respectivement, pour notre algorithme (voir section 2.3.3), pour l'algorithme décrit à la section 2.6.1.3, pour la variante de cet algorithme qui n'exécute pas l'étape 2e, pour la version "améliorée" de l'algorithme de la section 2.6.1.2 et, enfin, pour l'algorithme "de base" décrit dans cette section. Les temps fournis sont des moyennes calculées sur 100 exécutions successives, sauf pour le dernier algorithme, pour lequel seulement 10 exécutions ont été effectuées.

$\sharp\ eq$	t_{cs}	t_{ol}	$t_{ol'}$	t_{sh}	$t_{sh'}$
0	0,000	0,674	0,639	0,670	0,683
25.000	0,130	0,747	0,692	0,685	0,698
50.000	0,260	0,815	0,739	0,699	0,712
75.000	0,426	0,930	0,815	0,764	0,775
100.000	0,563	1,042	0,880	0,830	0,839
125.000	0,699	1,149	0,944	0,850	0,860
150.000	0,842	1,280	1,021	0,872	0,883
175.000	1,003	1,458	1,109	0,951	0,964
200.000	1,189	1,733	1,244	1,107	1,119
213.624	1,343	1,974	1,352	1,392	1,466
213.625	2,741	3,251	1,955	18,275	458,324
225.000	2,764	3,252	1,955	18,276	—
250.000	2,813	3,252	1,956	18,277	—

Table 2.8 – Variantes de l'algorithme de Shostak : Temps d'exécution

Les valeurs figurant dans la première ligne de la table s'expliquent par le fait que la collection de structures représentant l'ensemble de ter-

mes T est construite une fois pour toute avant toute exécution d'un des algorithmes de fermeture congruente, ce qui prend un temps approximativement égal à $0,67$ secondes. Par la suite, ces algorithmes manipulent les identifiants d'ensembles de structures représentant les termes. Notre algorithme ne construit pas les termes *a priori* mais étend la collection de structures "à la demande" pour qu'ils y soient représentés. Les temps d'exécution des cinq algorithmes sont assez semblables jusqu'à la résolution de la 213.625-ème équation. Nous verrons bientôt, en examinant le tableau 2.9, que cela s'explique du fait que la résolution de presque chacune des équations précédentes "ne pose pas de difficulté" et n'entraine qu'une modification locale de la collection de structure (notre algorithme) ou de la relation d'équivalence maintenue par la structure Union-Find (les algorithmes présentés dans cette section). Notre algorithme, parti avec un gros avantage de temps, perd petit à petit du terrain parce qu'il doit intégrer les termes constituant les équations à la collection de structure avant de disposer de leurs identifiants pour les unifier. Tous les algorithmes effectuent "le gros de leur travail" lors du traitement de la 213.625-ème équation. La version la plus simple de l'algorithme de Shostak ne peut en venir à bout et "sort de la mémoire" après 7 minutes et 38 secondes. La version optimisée de cet algorithme y parvient mais après un temps supérieur d'un ordre de grandeur à celui des trois premiers algorithmes. Notre algorithme prend un temps à peu près égal à la version du second algorithme qui "libère la mémoire" utilisée par les signatures "périmées". L'algorithme le plus rapide pour traiter cette équation est celui qui conserve toutes les structures utilisées pour les signatures. Après le traitement de cette équation, tous les termes de l'ensemble T sont égaux et, dans le cas de notre algorithme, la collection de structures représente en fait l'ensemble infini de tous les termes qu'on peut construire avec la constante a et le symbole de fonction f, tous ces termes étant considérés comme égaux. Les dernières équations sont donc "trivialement vraies" et traitées quasi instantanément par tous les algorithmes (sauf le dernier).

La table 2.9 présente les mesures faites sur l'utilisation de la mémoire par les différents algorithmes et, plus précisément, sur le nombre de structures utilisées à différentes étapes. On donne aussi, dans les deuxième et troisième colonnes, le nombre d'ensembles de structures (pour notre algorithme) et le nombre de classes d'équivalences (pour les différentes versions de l'algorithme de Shostak) [14]. Les versions de l'algorithme de Shostak présentées dans la section 2.6.1.2 ne modifient pas la collection

14. Le nombre (218.657), dans la troisième donne le nombre de classes d'équivalence restantes pour l'algorithme de Shostak "de base", lorsqu'il "sort de la mémoire". On voit qu'il est bien loin d'arriver au bout de son effort.

$\sharp\ eq$	$\sharp\ ids_{cs}$	$\sharp\ cls_{ol,sh}$	$\sharp\ str_{cs}$	$\sharp\ str_{ol}$	$\sharp\ str_{ol'}$
0	0	458.330	0	458.330	458.330
25.000	22.935	433.331	47.934	500.634	502.979
50.000	40.391	408.331	90.390	534.735	540.863
75.000	53.284	381.978	128.283	589.265	604.406
100.000	62.201	355.627	162.200	623.434	654.377
125.000	67.826	330.627	192.825	658.930	704.290
150.000	70.068	305.628	220.066	692.114	764.995
175.000	69.054	279.279	244.052	724.250	832.391
200.000	64.840	251.587	264.838	755.465	938.634
213.624	59.918	232.603	273.540	768.393	1.021.688
213.625	1	1(218.657)	2	458.331	1.087.013
225.000	1	1	2	458.331	1.087.013
250.000	1	1	2	458.331	1.087.013

Table 2.9 – Variantes de l'algorithme de Shostak : Utilisation de la mémoire

de structures initiale, puisqu'elles ne mémorisent pas les signatures, elles utilisent donc constamment 458.330 structures. On voit, dans la colonne $\sharp\ str_{cs}$ que notre algorithme utilise beaucoup moins de structures que les deux variantes de l'algorithme de la section 2.6.1.3 (colonnes $\sharp\ str_{ol}$ et $\sharp\ str_{ol'}$) : au pire, à peu près trois fois moins que l'algorithme qui libère la mémoire et quatre fois moins que celui qui ne la libère pas. À la fin de l'exécution, notre algorithme n'utilise plus que 2 structures au lieu de 458.331 et 1.087.013, respectivement, pour les deux autres.

Il est également intéressant d'analyser la quantité de mémoire nécessaire pour implémenter l'ensemble P, utilisé par les différentes versions de l'algorithme de Shostak et contenant les paires de termes à unifier (à placer dans la même classe d'équivalence). La table 2.10 indique le nombre maximum de paires contenues dans cet ensemble, implémenté par une pile, à différentes étapes d'exécution des algorithmes. (La paire initiale $\langle s, t\rangle$ n'est pas prise en compte.) Les valeurs indiquées dans une ligne donnée correspondent au maximum des valeurs obtenues pour chacune des équations traitées depuis la ligne précédente (25.000 équations, en général, sauf dans le cas particulier de l'équation 213.625). On voit que très peu de paires sont placées dans la pile dans le corps de la boucle des algorithmes : aucune en général, les valeurs non nulles étant le fait de très peu d'équations. La seule exception se produit pour l'équation 213.625, pour laquelle tous les algorithmes génèrent un très grand nombre de paires. On observe que, pour les algorithmes sauf le dernier, ce nombre

$\sharp$ *eq*	$\|P\|_{ol}$	$\|P\|_{ol'}$	$\|P\|_{sh}$	$\|P\|_{sh'}$
25.000	0	0	0	0
50.000	0	0	0	0
75.000	1.353	1.353	1.354	1.355
100.000	1.353	1.353	1.356	1.359
125.000	4	4	0	0
150.000	4	4	0	0
175.000	1.353	1.353	1.358	1.363
200.000	1.353	1.353	1.362	1.371
213.624	1.540	1.540	1.566	2.778
213.625	267.797	263.948	265.991	56.063.125
225.000	0	0	0	—
250.000	0	0	0	—

Table 2.10 – Variantes de l'algorithme de Shostak : Taille de la pile

de paires est à peine supérieur au nombre de classes d'équivalence restant à fusionner avant le traitement de cette équation, soit 232.603 (voir la table 2.9). Donc, les trois premiers algorithmes sont presque optimaux en ce que presque toutes les paires ajoutées à l'ensemble P donnent lieu à la fusion de deux classes d'équivalence. La différence d'efficacité entre l'algorithme optimisé de la section 2.6.1.2 et l'algorithme de la section 2.6.1.3, qui mémorise les signatures, est donc essentiellement due à l'utilisation de deux boucles imbriquées dans le premier des deux, contre une simple boucle, dans le second. Au contraire, l'algorithme simple de la section 2.6.1.2 engendre un très grand nombre de paires de termes ayant même signatures avec beaucoup de redondances entre elles et passe beaucoup plus de temps à créer ces paires qu'à les fusionner. Le nombre 56.063.125 de paires est plus de 200 fois plus élevé que le nombre obtenu pour les autres algorithmes et est cependant encore beaucoup trop petit pour que l'algorithme atteigne son but, puisque, à ce moment, il reste encore 218.657 classes à fusionner (seulement 13.946 l'ont été).

2.6.1.5 Synthèse

Nous pouvons maintenant synthétiser les similitudes et les différences entre notre méthode de résolution d'équations entre termes et l'algorithme de Shostak (dans son principe, indépendamment d'une variante spécifique). Après traitement d'un nombre quelconque d'équations, notre algorithme a construit une collection de structures E, utilisant un en-

semble d'identifiants I. Cette collection représente (dénote) une famille d'ensembles de termes $(T_i)_{i \in I}$. Chaque terme $f(t_1, t_2)$ figurant dans une équation déjà traitée est représenté, dans la collection, par une et une seule structure $f(i_1, i_2) : i$ qui est telle que $f(t_1, t_2) \in T_i$, $t_1 \in T_{i_1}$ et $t_2 \in T_{i_2}$. Dans l'algorithme de Shostak, les termes des équations déjà traitées sont répartis en classes d'équivalences maintenues par la structure Union-Find. Il y a une correspondance biunivoque entre l'ensemble de ces classes d'équivalences et l'ensemble d'identifiants I et donc aussi entre l'ensemble I et l'ensemble des représentants de ces classes. Du coup, il y a une correspondance biunivoque entre les structures de la collection E et les signatures des termes des équations déjà traitées. Si nous notons s_i le représentant de la classe d'équivalence associée à l'identifiant i, il correspond à la structure $f(i_1, i_2) : i$, la signature $f(s_{i_1}, s_{i_2})$. Lors du traitement d'une équation particulière, notre algorithme maintient, pour chaque clé $f(i_1, i_2)$, une liste des structures qui ont cette clé afin d'"unifier" leurs identifiants. L'algorithme de Shostak, maintient, à la place une liste de paires de termes ayant même signature, pour fusionner leurs classes d'équivalence.

Le principal avantage de notre méthode sur celle de Shostak est limiter l'utilisation de la mémoire en conservant une seule structure pour tous les termes ayant même signature et pour cette signature elle-même. Cela nous permet aussi de manipuler des ensembles infinis de termes si l'ensemble de leurs classes d'équivalence est fini. Au point de vue du temps d'exécution, notre méthode est compétitive avec toutes les versions de l'algorithme de Shostak.

2.6.1.6 Autres algorithmes de fermeture congruente

Nous comparons maintenant brièvement notre travail à deux références incontournables. Un point commun à ces deux articles est qu'ils considèrent le problème de calculer la fermeture congruente d'une relation définie sur les nœuds d'un graphe, non sur un ensemble de termes. La plus grande généralité ainsi obtenue rend un peu plus difficile la tâche de mettre notre travail en perspective par rapport à ces travaux.

L'algorithme de Nelson et Oppen, décrit dans [NO80, BM07], est, en fait, très semblable à l'algorithme que nous avons décrit à la section 2.6.1.2. La notion de terme utilisée dans l'algorithme de Shostak est remplacée par celle de sommet d'un graphe, les successeurs d'un sommet jouant le rôle des sous-termes stricts. Au lieu d'utiliser les listes $\mathit{liste}(\overline{t})$ (des termes dont la signature contient $\overline{t}$, comme sous-terme direct), l'algorithme utilise la liste des prédécesseurs des sommets équivalents à un sommet

donné. La complexité de l'algorithme est en $O(m^2)$ où m est le nombre d'arcs du graphe.

L'algorithme de Downey, Sethi et Tarjan [DST80] s'applique également à une relation sur un graphe. Les auteurs insistent sur le fait que le graphe ne doit pas nécessairement être acyclique pour garantir la correction de leur algorithme. Cette plus grande généralité permet d'appliquer leur algorithme non seulement à une relation entre termes, mais aussi à une collection de structures, puisque qu'il est clair qu'une collection de structures équivaut à un graphe dont les sommets sont les identifiants d'ensembles de structures et les arcs sont les structures (une structure représente en fait deux arcs). Avec cette interprétation, nous pouvons donner un algorithme très semblable au leur. Soit E une collection de structures utilisant l'ensemble d'identifiants I. Soit ρ, une relation sur I. Nous cherchons la plus petite relation d'équivalence sur I qui contient ρ et qui est compatible avec les structures de E. C'est-à-dire qu'on a :

$$\left.\begin{array}{c} i_k \equiv j_k \ (i = 1, 2) \\ f(i_1, i_2) : i \in E \\ f(j_1, j_2) : j \in E \end{array}\right\} \implies i \equiv j$$

pour tous les identifiants i, j, i_1, j_1, i_2, $j_2 \in I$. Notre version de leur algorithme, adapté pour les collections de structures, est donnée ci-dessous.

Notons E_0, la collection de structures initiale à laquelle on applique l'algorithme et soit I_0 son ensemble d'identifiants. Soit ρ, une relation binaire sur I_0. On peut la voir comme une liste d'"équations" entre des éléments de I_0. L'algorithme calcule la fermeture congruente de ρ sous forme d'une structure Union-Find contenant ses classes d'équivalence. On note $\overline{i}$ le représentant de la classe d'équivalence courante de i, à un moment donné (pour tout $i \in I_0$). L'algorithme modifie simultanément la collection de structures, à partir de sa valeur initiale, et la structure Union-Find. On a donc, après traitement d'un nombre quelconque de paires d'élément de ρ, une collection de structures courante E utilisant l'ensemble d'identifiants courant I. Cet ensemble I est égal à l'ensemble des représentants des classes d'équivalence mémorisées dans la structure Union-Find. Il y a donc double emploi entre la collection de structures et la structure Union-Find mais cette redondance est nécessaire pour faire correspondre aux paires $\langle i, j\rangle$ de la relation ρ, leurs paires courantes de représentants $\langle \overline{i}, \overline{j}\rangle$. Par contre, notre version de l'algorithme ne conserve pas une copie de la collection de structures initiale.

Comme nous l'avons fait précédemment, nous donnons une version de l'algorithme qui traite une seule paire d'identifiants $\langle k, l\rangle$. L'algorithme complet applique cet algorithme partiel, successivement, à chaque paire

de la relation ρ.

1. Si $\overline{k} = \overline{l}$, l'algorithme se termine sans rien faire.
2. Sinon, supposons que l'identifiant $\overline{k}$ figure dans au moins autant de structures que l'identifiant $\overline{l}$ (on échange les rôles de k et l sinon). On exécute, dans l'ordre :
   (a) appliquer l'opération *substitute* [15] aux identifiants $\overline{k}$ et $\overline{l}$;
   (b) fusionner les classes de $\overline{k}$ et $\overline{l}$, en choisissant $\overline{k}$ comme représentant.
3. Ensuite, tant que la collection de structure courante contient deux structures $key : i$, $key : j$, distinctes mais de même clé, on exécute (en supposant que l'identifiant i figure dans au moins autant de structures que j) :
   (a) appliquer l'opération *substitute* aux identifiants i et j ;
   (b) fusionner les classes de i et j, en choisissant i comme représentant.

$\sharp\ eq$	$\sharp\ ids_{cs}$	$\sharp\ ids/cls_{dst}$	$\sharp\ str_{cs}$	$\sharp\ str_{dst}$	t_{cs}	t_{dst}
0	0	458.330	0	458.330	0,000	0,713
25.000	22.935	433.331	47.934	458.330	0,130	0,782
50.000	40.391	408.331	90.390	458.330	0,260	0,852
75.000	53.284	381.978	128.283	456.977	0,426	0,930
100.000	62.201	355.627	162.200	455.626	0,563	1,011
125.000	67.826	330.627	192.825	455.626	0,699	1,090
150.000	70.068	305.628	220.066	455.626	0,842	1,177
175.000	69.054	279.279	244.052	454.277	1,003	1,281
200.000	64.840	251.587	264.838	451.585	1,189	1,412
213.624	59.918	232.603	273.540	446.225	1,343	1,542
213.625	1	1	2	2	2,741	3,070
225.000	1	1	2	2	2,764	3,072
250.000	1	1	2	2	2,813	3,075

Table 2.11 – Comparaison avec l'algorithme de Downey, Sethi et Tarjan

Ce dernier algorithme est plus général que ceux vus précédemment car il permet de "résoudre" un ensemble d'équations entre termes par rapport à une collection de structures donnée quelconque, pas seulement une collection de structures représentant un ensemble de termes tous non

15. voir section 2.3.2.2

équivalents. On peut cependant, et évidemment, aussi appliquer l'algorithme à une collection de structures (de départ) représentant des termes non équivalents. La différence avec notre algorithme de résolution d'un ensemble d'équations entre termes (voir section 2.5.6) est qu'on calcule la représentation de tous les termes au départ et qu'on utilise la structure Union-Find pour faire correspondre, à tout moment, les identifiants de départ des termes, à l'identifiant qui représente leur classe d'équivalence. Cette méthode est plus rapide pour déterminer les identifiants à unifier à chaque étape, car elle n'utilise pas l'opération *toSet* (voir section 2.3.2.1) pour mettre les termes en correspondance avec leurs identifiants. Par contre, l'algorithme obtenu n'est pas incrémental : il faut, avant toute autre chose, construire tous les termes pour déterminer l'ensemble de leurs identifiants afin de pouvoir calculer ensuite la fermeture congruente au moyen de la structure Union-Find.

Nous concluons cette comparaison par une mise en correspondance des performances de notre algorithme de résolution d'un ensemble d'équations entre termes avec (notre adaptation de) l'algorithme de Downey, Sethi et Tarjan. Nous utilisons les mêmes données de test qu'à la section 2.6.1.4. Les résultats sont montrés dans la table 2.11. On constate à nouveau que notre algorithme utilise moins de mémoire, vu son caractère incrémental. Les temps d'exécution sont comparables avec également un léger avantage à notre algorithme. (Nous supposons que le lecteur saura interpréter les résultats présentés en adaptant les indications données pour les tables 2.8 à 2.10).

2.6.2 Représentation des termes et des ensembles de termes

"Calculer avec des termes" est essentiel dans de nombreux domaines de l'informatique : démonstrateurs de théorèmes, langages de programmation logiques et fonctionnels, bases de données "déductives", par exemple. De nombreuses méthodes de représentation de termes ou d'ensembles de termes ont donc été proposées pour rendre ce calcul efficace dans chaque domaine spécifique.

2.6.2.1 Indexation des ensembles de termes

Dans chacun des domaines cités dans l'introduction de la section 2.6.2, des méthodes de représentation de termes et d'ensembles de termes ont été proposées pour résoudre efficacement des problèmes du genre : étant donné un ensemble T de termes et un terme particulier t, déterminer tous les termes $l \in T$ qui sont dans une relation R avec t (on a : $l\,R\,t$).

Le terme t est souvent appelé *la question*. L'ensemble de termes T doit quant à lui être représenté de manière à "répondre à la question" (trouver l'ensemble des termes l) aussi efficacement que possible. On dit alors que les termes de T sont *indexés* et on appelle T, ou plutôt sa représentation, *l'index* [SRV01]. Selon le domaine d'application, des choix fréquents pour la relation $l\,R\,t$ peuvent être que l est une instance, ou est une généralisation, ou est égal à t, ou est unifiable avec t.

Il est souvent nécessaire de représenter les termes de manière compacte pour limiter l'utilisation de la mémoire. Pour cela, on choisit souvent de représenter les termes non comme des arbres mais comme des graphes orientés acycliques (DAGs) où chaque sous-terme n'est représenté qu'une seule fois. On peut le faire localement lorsqu'un nouveau terme est créé ou maintenir une "base de termes" dans laquelle chaque terme n'est représenté qu'une seule fois. Si l'index est représenté de cette façon, on peut efficacement répondre à la question : $t \in T$? Pour répondre à d'autres sortes de questions telles que mentionnées ci-dessus, d'autres méthodes d'implémentation de l'index ont été proposées, basées sur l'utilisation d'automates ou d'arbres de suffixes, principalement. On pourra consulter à ce sujet la référence [SRV01] qui fournit aussi une bibliographie sur le sujet.

Notre implémentation des collections de structures (voir la section 2.4) est, en partie, analogue à une "base de termes" telle que définie ci-dessus. En particulier, dans le cas d'une collection de structures creuse (voir l'introduction de la section 2.5), chaque identifiant (d'ensemble de structures) représente un seul terme et ce terme n'est représenté qu'une fois. Dans le cas général, cependant, le même identifiant représente plusieurs termes, ce qui rend notre représentation plus compacte encore que les "bases de termes" classiques ; en particulier, nous pouvons représenter certains ensembles de termes infinis, ce que les "bases de termes" classiques ne peuvent faire.

Au niveau des domaines d'application, une grande différence existence entre nos collections de structures et les méthodes décrites dans [SRV01] et évoquées ci-dessus. Ces méthodes servent à représenter des ensembles finis de termes *a priori* quelconques, définis, en quelque sorte, par simple énumération, alors que nos collections de structures représentent des ensembles de termes égaux dans une certaine théorie, par exemple équationnelle mais pas nécessairement. Il s'en suit que la réponse à la "requête" : trouver tous les termes l égaux à t, est constitué d'un ensemble "non trivial" de termes, au contraire des applications classiques basées sur l'égalité "syntaxique". Parmi les autres applications possibles de nos collections, il y a bien sûr : trouver un terme simplifié égal à t, ou, en

d'autres mots : simplifier t. Nous le faisons en extrayant de l'ensemble des termes égaux à t, un ou plusieurs termes les plus "simples" selon tel ou tel critère donné.

Notre affirmation que les "bases de termes" et, plus généralement, les méthodes d'indexation classiques décrites dans [SRV01] ne permettent de représenter que des ensembles finis de termes doit être relativisée en remarquant que ces méthodes prennent en compte des termes contenant des variables, ce que nos collections de structures n'autorisent pas, strictement parlant, bien que nous autorisions l'utilisation de constantes "non interprétées" (voir le point 2. de la section 2.3.3.1). Les deux approches sont finalement, en quelque sorte, "incommensurables".

Les méthodes décrites dans [SRV01], basées sur l'égalité "syntaxique" des termes, ont été généralisées à certaines propriétés équationnelles générales telles que l'associativité et la commutativité, donnant lieu à des méthodes spécialisées pour ces propriétés. Notre approche au contraire est uniformément applicable à n'importe quelle théorie égalitaire ou à des théories plus générales telles que celle des expressions rationnelles. En contrepartie, nous ne représentons pas dans les collections de structures, des termes avec des variables.

2.6.2.2 Automates d'arbres

La notion de collection de structures, définie et étudiée dans ce chapitre, a beaucoup de points communs avec la notion d'automate d'arbres [TW68, CDG$^+$07]. Un automate d'arbres est un objet mathématique qui spécifie un algorithme de reconnaissance d'un ensemble de termes. Cet algorithme décide si un terme donné appartient à l'ensemble défini par l'automate d'arbres. Chaque structure $f(i_1, i_2) : i$, d'une collection de structures correspond dans un automate d'arbres à une règle de transition. L'application des règles de transition permet d'associer à chaque terme un état qui correspond, dans les collections de structures, à un identifiant. L'ensemble de termes reconnus par l'automate est formé de tous les termes qui sont associés par l'application des règles à un état *acceptant*.

Les collections de structures peuvent donc être vues comme une version simplifiée des automates d'arbres puisqu'elles n'utilisent pas la notion d'état acceptant. La sémantique des collections de structures est aussi définie différemment : par une famille d'ensembles de termes ou par une relation entre termes. Ces différences ne sont pas essentielles. La différence la plus importante entre notre travail et les travaux relatifs aux automates se situe au niveau des opérations que nous avons définies pour

manipuler les collections de structures ainsi que dans la façon dont nous implémentons les collections de structures pour rendre ces opérations efficaces. C'est certainement le cas qu'il existe de nombreux problèmes théoriques ou pratiques ou il est utile de combiner les opérations que nous avons définies sur les collections de structures à d'autres opérations classiquement définies sur les automates d'arbres. Nous en donnons d'ailleurs un exemple à la section 2.6.3.2. Néanmoins, nous pensons qu'il ne serait pas utile de reformuler la "théorie" des collections de structures, élaborée à la section 2.3, en termes de théorie des automates d'arbres car cela n'apporterait pas un meilleur éclairage sur l'utilité et les propriétés intéressantes des collections de structures.

2.6.3 Autres domaines de recherche

2.6.3.1 Règles de réécriture

La notion de règle de réécriture [DJ90, Dic91] a été initialement proposée dans [KB70]. Une règle de réécriture est une équation entre termes qui est "orientée" et que l'on écrit $t \rightarrow s$ où s et t sont des termes. Elle signifie intuitivement qu'on peut "simplifier" un terme quelconque en remplaçant un de ses sous-termes égal à t (ou à une instance de t) par s (ou par l'instance correspondante de s). Étant donné un ensemble d'équations entre termes Eq, on est intéressé à trouver un ensemble de règles de réécriture R telles que l'application répétée des règles à un terme quelconque produit un terme qui n'est pas simplifiable, appelé *terme canonique*. Ce terme doit être le même quel que soit l'ordre d'application des règles. Le fait pour deux termes d'être simplifiables dans le même terme canonique fournit une condition suffisante pour l'égalité de ces deux termes (par rapport aux modèles des équations de Eq). On est particulièrement intéressé par les ensembles de règles pour lesquels cette condition est aussi nécessaire puisque ces ensembles de règles fournissent un algorithme de décision pour le problème de l'égalité de deux termes.

Dans les démonstrateurs de théorèmes basés sur l'idée de faire coopérer différents algorithmes de décision [NO79], on se restreint souvent aux règles de réécriture n'utilisant que des termes clos [NO05] et on utilise ces ensembles de règles pour "passer de l'information" d'un composant du démonstrateur de théorèmes à un autre. Les algorithmes de fermeture congruente (voir section 2.6.1) sont souvent utilisés pour construire de tels ensembles de règles de réécriture sur base de la fermeture congruente [Sny93]. Une des difficultés est d'obtenir un ensemble de règles *réduit* [Sny93], dans lequel un terme canonique ne contient pas de sous-terme égal à la partie gauche d'une règle.

Il est immédiat de vérifier que l'utilisation de notre algorithme de résolution d'un ensemble d'équations entre termes fournit "trivialement" un ensemble de règles de réécriture possédant toutes les qualités mentionnées ci-dessus. Il suffit pour cela d'associer à chaque identifiant i de la collection de structures, résultant de la résolution des équations, une "nouvelle constante" c_i. L'ensemble des règles de réécriture est formée de toutes les règles de la forme $f(c_{i_1}, c_{i_1}) \rightarrow c_i$, où $f(i_1, i_2) : i$ est une structure de la collection. L'application des règles à un terme quelconque équivaut à lui appliquer l'opération *toSet* (voir section 2.3.2.1). Vérifier l'égalité de deux termes par application des règles de réécriture équivaut à leur appliquer l'opération *toSet* puis à tester si leurs identifiants sont égaux (voir section 2.3.3). Pour des raisons techniques ou esthétiques, il se peut qu'on désire produire un ensemble de règles de réécriture n'incorporant pas de nouvelle constante. On peut alors remplacer les constantes c_i par un terme t_i choisi dans chaque ensemble de termes T_i associé à l'identifiant i. On peut choisir un terme de taille minimale et éviter d'incorporer la règle $t_i \rightarrow t_i$, à l'ensemble des règles.

2.6.3.2 Unification rigide

Les collections de structures sont applicables à un problème plus général que la résolution d'équations entre termes clos : l'unification dite *rigide* par rapport à un ensemble fini d'équations entre termes contenant des *variables* [GRS87]. Nous n'avons pas, à ce jour, implémenté concrètement d'algorithme pour résoudre ce type de problème mais nous donnons ci-dessous le principe de la méthode à utiliser.

Le problème *simple* de l'unification rigide s'énonce comme suit. Étant donné un ensemble Eq d'équations entre termes non nécessairement clos, et une équation entre termes $u = v$, décider s'il existe une instanciation des variables utilisées par ces équations telle que les termes u et v (instanciés) soient égaux dans tous les modèles des équations Eq (instanciées). Le problème *général* de l'unification rigide considère plusieurs problèmes simples en même temps et recherche s'il existe une instanciation commune qui les résout tous simultanément.

On peut résoudre le problème simple, au moyen des collections de structures, comme suit. Nous nous limitons à une seule variable, pour commencer.

1. On remplace la variable par une "nouvelle" constante c dans les différentes équations. (Il s'agit essentiellement d'un "changement de point de vue" sur la variable concernée.)
2. On construit la collection de structures définie par la résolution

des équations Eq (modifiées, donc closes). On l'étend ensuite, en appliquant l'opération *toSet* aux termes u et v (modifiés), *sans unifier les identifiants obtenus.*

3. Si les identifiants correspondant aux termes u et v sont égaux, n'importe quelle instanciation de la variable résout le problème. La réponse au problème est positive.
4. Sinon, soit i_c, l'identifiant correspondant à la constante c dans la collection de structures. On considère tous les identifiants i de la collection de structures dont l'ensemble de termes associé T_i contient au moins un terme qui ne contient pas la constante c (donc $i \neq i_c$).
   (a) On applique l'opération *unify* aux identifiants i et i_c.
   (b) Si les termes u et v ont même identifiant dans la collection de structures ainsi obtenue, le problème a une solution et tous les termes de l'ensemble $T'_{i'}$ où i' est l'identifiant correspondant à i et i_c, dans la collection de structures modifiée, fournissent une telle solution (en fait, ceux qui ne contiennent pas c, il y en a au moins un).
   (c) Sinon, on restore la collection de structures obtenue au point 2, et on reprend au point 4a, avec un autre identifiant i.
5. Si aucun des identifiants i ne fournit une solution, aucune solution n'existe.

Nous pouvons justifier la correction de cet algorithme, comme suit, en utilisant les propriétés des collections de structures démontrées dans la section 2.3.

1. Il est évident que l'algorithme est correct s'il se termine au point 3. Nous supposons ci-dessous qu'il ne s'y termine pas.
2. Un terme t résout le problème s'il ne contient pas la variable c et si les termes u et v sont égaux dans tous les modèles de $Eq \cup \{c = t\}$. Ce qui revient à dire que les termes u et v sont représentés par le même identifiant dans la collection de structures obtenue en résolvant l'ensemble d'équations $Eq \cup \{c = t\}$.
3. Si un terme t résout la question, il doit nécessairement être représenté dans la collection de structures obtenue au point 2 de l'algorithme, car, si ce n'était pas le cas, le fait de lui appliquer l'opération *toSet* ne ferait qu'ajouter de nouvelles structures à la collection de structures, sans modifier les ensembles de structures E_i préexistants. Ensuite, le fait de modifier la collection en résolvant l'équation $c = t$ ne ferait que placer la structure créée

pour remplacer t dans l'ensemble de structures E_{i_c}, sans entraîner d'autre modification de la collection de structures. Donc, les identifiants de u et v, distincts au départ, le resteraient.

La méthode expliquée ci-dessus est plus satisfaisante que celle expliquée dans [DGN+98] car elle fournit un algorithme précis, alors que les auteurs de [DGN+98] se contentent de prouver qu'un tel algorithme existe en combinant des résultats d'autres auteurs. De plus, notre justification est plus simple grâce à l'utilisation des collections de structures et l'exploitation des propriétés de leurs opérations. Enfin, notre algorithme fournit une description de l'ensemble de toutes les solutions, c'est-à-dire l'équivalent d'un *unificateur général*.

On peut généraliser la méthode précédente à plus d'une variable en procédant par essais et erreurs et en affectant successivement des valeurs aux variables (considérées comme des constantes). Par contre, le problème général de l'unification rigide simultanée n'est soluble que pour une seule variable (voir [DGN+98]). On peut le résoudre en calculant les collections de structures correspondant aux solutions de chaque problème simple, puis en vérifiant si elles contiennent des solutions communes. Pour cela, on peut interpréter les collections de structures comme des automates d'arbres avec pour état acceptant l'identifiant i_c et calculer l'automate qui reconnaît l'ensemble de termes égal à l'intersection des ensembles de termes correspondant à i_c dans les différentes collections de structures.

Si l'ensemble obtenu est non vide, tous les termes de cet ensemble fournissent une solution au problème.

Chapitre 3

Simplification des expressions avec les collections de structures

Dans ce troisième et dernier chapitre, nous appliquons enfin les collections de structures au problème de la simplification des expressions. Dans la section 3.1, nous donnons un algorithme efficace pour calculer et maintenir incrémentalement la taille minimale des termes contenus dans chaque classe d'équivalence de la congruence représentée par une collection de structures. Sur base de cette taille, on calcule facilement les termes minimaux, eux-mêmes. Dans la section 3.2, nous décrivons un autre algorithme qui permet de calculer la collection de structures représentant la congruence définie par un ensemble fini d'équations *non closes* et un ensemble fini de constantes, lorsque le nombre de classes d'équivalences de cette congruence est fini. Nous utilisons cet algorithme pour minimiser les formules booléennes utilisant au plus trois lettres distinctes et nous discutons sur cette base l'applicabilité de l'algorithme, en général. Enfin, dans la section 3.3, nous présentons une approche pour résoudre le problème à l'origine de tout ce travail : la simplification des expressions. Nous remplaçons l'approche de la section 3.2, par une approche "guidée par l'objectif" conduisant à des algorithmes qui créent des expressions égales, ou "susceptibles de devenir égales" aux sous-expressions d'une expression à simplifier de départ. Lorsque la taille de l'expression diminue, ces algorithmes "recentrent la recherche" sur les structures de l'expression réduite. Plusieurs stratégies et heuristiques sont proposées. Nous appliquons les algorithmes aux formules booléennes et évaluons leurs performances pour des formules de 800 symboles utilisant jusqu'à 20 lettres distinctes.

3.1 Calcul des termes de taille minimale

Étant donnée une collection de structures E et un terme t représenté dans E, il est facile de déterminer les termes équivalents à t les plus simples selon différents critères, pourvu que ces critères s'énoncent facilement sur base de la structure des termes. Dans cette section, nous expliquons comment le faire en utilisant comme critère la taille des termes : nous cherchons les termes utilisant le moins de symboles possibles, équivalents à un terme donné.

3.1.1 Calcul des tailles minimales

3.1.1.1 Théorie du problème

La taille d'un terme t, notée $|\,t\,|$, est le nombre de ses symboles (en notation polonaise), compte non tenu des occurrences du symbole *null*. On a donc :

$$\begin{aligned} |\,null\,| &= 0, \\ |\,f(t_1,t_2)\,| &= |\,t_1\,| + |\,t_2\,| + 1. \end{aligned}$$

Soit E, une collection de structures bien formée mais pas nécessairement normalisée. Soit $(T_i)_{i\in I}$, sa dénotation concrète. Chacun des ensembles T_i est non vide et possède un ou plusieurs termes de taille minimale. On définit, pour tout identifiant $i \in I$:

$$\text{size}(i) \;=\; \min\{\, |\,t\,| \;\mid\; t \in T_i\}.$$

Le symbole size représente donc une fonction de I dans $\mathbb{N}$. Nous allons donner un algorithme pour calculer cette fonction, basé sur le fait qu'elle est l'unique "point fixe" d'une transformation de fonction définie ci-dessous. Nous définissons d'abord, pour toute fonction $ta : I \longrightarrow \mathbb{N}$:

$$\begin{aligned} \text{relSize}\,(null,\ ta) &= 0, \\ \text{relSize}\,(f(i_1,i_2),\ ta) &= ta(i_1) + ta(i_2) + 1. \end{aligned}$$

Nous définissons ensuite la transformation de fonction trSize : $(I \longrightarrow \mathbb{N}) \longrightarrow (I \longrightarrow \mathbb{N})$, comme suit :

$$\text{trSize}\,(ta)(i) \;=\; \min\{\, \text{relSize}\,(f(i_1,i_2),\ ta) \;\mid\; f(i_1,i_2) : i \in E \,\}.$$

On dit qu'une fonction ta est un point fixe de la transformation trSize si

$$\text{trSize}\,(ta) \;=\; ta.$$

Nous allons maintenant démontrer que la fonction size est l'unique point fixe de la transformation trSize. Ceci nous permettra de calculer la fonction size par un "algorithme de point fixe" opérant sur la collection de structures E.

Preuve que la fonction size est l'unique point fixe de la transformation trSize.

1. Supposons que la transformation trSize possède un point fixe ta. Nous pouvons ranger les identifiants de l'ensemble I selon une suite $i_1, i_2, \ldots, i_n$, de telle sorte qu'on ait : $ta(i_1) \leq ta(i_2) \leq \ldots \leq ta(i_n)$. Nous montrons, par récurrence sur l, qu'on a : $\text{size}(i_l) \leq ta(i_l)$ pour tout l tel que $1 \leq l \leq n$.
   C'est vrai, pour $l = 1$ car, dans ce cas : $\text{size}(i_1) = \text{size}(i_{null}) = 0 = ta(i_1)$.
   Supposons ensuite que $l > 1$. Dans ce cas, il existe une structure $f(i_j, i_k) : i_l$, appartenant à E telle que $i_j, i_k < i_l$ et $\text{relSize}\,(f(i_j, i_k), ta) = ta(i_l)$. En effet, pour toutes les structures $f(i_j, i_k) : i_l$ telles que $i_j \geq i_l$ ou $i_k \geq i_l$, on a : $\text{relSize}\,(f(i_j, i_k), ta) \geq ta(i_l) + 1$. Pour cette structure telle que $i_j, i_k < i_l$, on a : $\text{size}(i_l) \leq \text{size}(i_j) + \text{size}(i_k) + 1 \leq ta(i_j) + ta(i_k) + 1 = ta(i_l)$. D'où $\text{size}(i_l) \leq ta(i_l)$. (On a appliqué l'hypothèse de récurrence sur i_j et i_k.)
2. Nous montrons ensuite, par induction sur la structure des termes, que pour tout identifiant $i \in I$ et pour tout terme $t \in T_i : ta(i) \leq |\, t \,|$.
   C'est évidemment vrai pour l'identifiant i_{null}.
   Sinon t s'écrit sous la forme $f(t_1, t_2)$ et il existe une structure $f(i_1, i_2) : i$ appartenant à E telle que $t_j \in T_{i_j}$ $(j = 1, 2)$. Par hypothèse d'induction, on a : $ta(i_j) \leq |\, t_j \,|$ $(j = 1, 2)$. Donc, $ta(i) \leq ta(i_1) + ta(i_2) + 1 \leq |\, t_1 \,| + |\, t_2 \,| + 1 = |\, t \,|$.
3. Le point 2, ci-dessus, implique évidemment, pour tout $i \in I$, que $ta(i) \leq \text{size}(i)$, puisque les ensembles T_i sont non vides. En combinant ce résultat avec celui du point 1, on trouve que les fonctions ta et size sont égales. □

Note 13. La démonstration précédente n'est en réalité pas tout à fait complète car nous n'avons pas démontré que la transformation de fonction trSize a au moins un point fixe. L'algorithme que nous donnons un peu plus loin en construit un. Son existence sera ainsi établie.

3.1.1.2 Algorithme de calcul des tailles minimales

Nous donnons maintenant un algorithme pour calculer le point fixe de la transformation trSize associée à la collection de structures E. Outre la

collection de structures, cet algorithme utilise un tableau d'entiers *tsize*, indicé par les éléments de I et une variable L contenant un ensemble (variable) d'identifiants appartenant à I.

L'algorithme est construit et justifié sur base de l'invariant suivant.

(a) $\forall i \in I \ : \ tsize(i) \geq \text{trSize}(tsize)(i)$,
(b) $\forall f(i_1, i_2) : i \in E \ : \ i_1, i_2 \notin L \implies \text{relSize}(f(i_1, i_2), tsize) \geq tsize(i)$

Voici l'algorithme.

1. Pour tout identifiant $i \in I \setminus \{i_{null}\}$, on exécute $tsize[i] := \infty$. On fait aussi $tsize[i_{null}] := 0$ et $L := \{i_{null}\}$.
2. Tant que $L \neq \{\}$, on effectue ce qui suit.
   (a) On prélève un identifiant j dans L.
   (b) Pour toutes les structures $f(i_1, i_2) : i \in E$ telles que $j \in \{i_1, i_2\}$ et $\text{relSize}(f(i_1, i_2), tsize) < tsize[i]$, on effectue $tsize[i] := \text{relSize}(f(i_1, i_2), tsize)$ et on ajoute i à L, s'il n'y appartenait pas déjà.

Justifions la correction de l'algorithme.

Preuve du fait que l'algorithme calcule le point fixe de trSize.

1. L'algorithme se termine car un élément est retiré de L à chaque itération et, dans le cas où des identifiants sont ajoutés à L, un ou plusieurs éléments du tableau *tsize* sont diminués. Comme tous les éléments de *tsize* sont bornés inférieurement par 0, ils ne peuvent être diminués qu'un nombre fini de fois. Donc, après un certain nombre d'itérations, le tableau *tsize* ne sera plus modifié. Dès lors, la variable L sera amputée d'un identifiant à chaque itération et l'algorithme se terminera.
2. Convenons que $\text{relSize}(f(i_1, i_2), tsize) = \infty$ lorsque $tsize[i_1] = \infty$ ou $tsize[i_2] = \infty$. Dès lors l'invariant est vérifié après l'étape 1.
3. En vue de vérifier le respect de l'invariant à chaque itération, remarquons d'abord que l'identifiant j ne peut être replacé dans L à l'étape 2b, puisque, si c'était le cas, on devrait avoir $j \in \{i_1, i_2\}$ et $i = j$. Or il est impossible d'avoir $\text{relSize}(f(i_1, i_2), tsize) < tsize[j]$, si $j \in \{i_1, i_2\}$. Cela implique aussi que la valeur $tsize[j]$ reste inchangée.
4. Le point (a) de l'invariant est maintenu à chaque itération. Il faut le vérifier seulement pour les identifiants i tels que $tsize[i]$ est diminué puisque $\text{trSize}(tsize)(i)$ ne peut que diminuer. Lorsqu'une instruction $tsize[i] := \text{relSize}(f(i_1, i_2), tsize)$ est exécutée,

nécessairement : $i \notin \{i_1, i_2\}$. Donc, la valeur de relSize($f(i_1, i_2)$, *tsize*) n'est pas changée. Par conséquent, on obtient : $tsize[i] = \text{relSize}(f(i_1, i_2), tsize) \geq \text{trSize}(tsize)(i)$.

5. Le point (b) de l'invariant est aussi maintenu à chaque itération. Soient i_1 et i_2, deux identifiants tels que $i_1, i_2 \notin L$ *après* une itération. Alors, aucune des valeurs $tsize[i_1]$, $tsize[i_2]$ n'a changé lors de l'itération. Montrons le pour i_1, par exemple. Ou bien, i_1 n'appartenait pas à L, avant l'itération. Alors, $tsize[i_1]$ n'a pas changé sinon i_1 aurait été placé dans L. Ou bien, au contraire, i_1 appartenait à L. Alors, on a nécessairement $i_1 = j$. Or, comme montré plus haut, $tsize[j]$ ne peut avoir diminué non plus. Puisque ni $tsize[i_1]$ ni $tsize[i_2]$ n'ont changé, la valeur de relSize($f(i_1, i_2)$, *tsize*) n'a pas changé lors de l'itération et la condition relSize($f(i_1, i_2)$, *tsize*) $\geq$ $tsize[i]$ reste donc vraie puisque $tsize[i]$ ne peut que diminuer.
6. Lorsque l'algorithme se termine, le point (b) de l'invariant implique qu'on a : $\forall f(i_1, i_2) : i \in E \ : \ \text{relSize}(f(i_1, i_2), tsize) \geq tsize[i]$, ce qui implique : $\forall i \in I \ : \ \text{trSize}(tsize)(i) \geq tsize[i]$. Cette condition, combinée au point (a), implique que *tsize* est un point fixe de la transformation trSize. On a donc aussi $tsize[i] = \text{size}(i)$ pour tout identifiant $i \in I$. □

3.1.1.3 Discussion

1. Il est immédiat d'implémenter l'algorithme présenté ci-dessus avec les *structures de données concrètes* que nous utilisons pour réaliser les collections de structures (voir la section 2.4). Le parcours des structures $f(i_1, i_2) : i$ telles que $j \in \{i_1, i_2\}$ est effectué optimalement en parcourant une seule fois les listes $sameId_1(j)$ et $sameId_2(j)$. La variable L contenant les identifiants j restant à considérer est implémentée avec un nouvelle instance de la classe `OneList`. Le tableau *tsize* est un simple tableau d'entiers de taille égale à `NBRSETS`.
2. Nous ne donnons pas d'*étude de complexité* précise de l'algorithme. Dans les applications des collections de structures que nous avons réalisées, le temps d'exécution dévolu à son exécution est toujours apparu négligeable par rapport au temps des autres opérations sur les collections de structures. L'algorithme est donc suffisamment efficace pour la pratique. Nous pouvons malgré tout donner une estimation approximative (optimiste) de sa complexité. Les identifiants j de la collection de structures sont placés dans L seulement lorsque que la valeur $tsize[j]$ diminue. La plupart

du temps, cela n'arrive qu'une fois car on considère d'abord les identifiants "de plus petite taille", de sorte que, quand la valeur relSize($f(i_1, i_2)$, *tsize*) diminue pour la première fois, il y a de forte chance qu'elle prenne sa valeur définitive parce que les éléments $tsize[i_1]$ et $tsize[i_2]$ ont déjà reçu la leur. Il s'en suit que chaque structure $f(i_1, i_2) : i$ n'est considérée que deux fois par l'algorithme : chaque fois que les nombres $tsize[i_1]$ et $tsize[i_2]$ reçoivent leur valeur. Sous cette hypothèse, la complexité de l'algorithme est en $O(N)$: linéaire sur le nombre de structures de la collection.

3. L'algorithme que nous proposons peut être utilisé de façon *incrémentale*, en conservant globalement l'instance de la classe `OneList` qui réalise la variable L, et le tableau *tsize*. Une (ré-)exécution de l'étape 2 de l'algorithme doit être envisagée seulement lorsqu'une opération de renommage d'une structure $f(i_1, i_2) : i$ en $f(i'_1, i'_2) : i'$ diminue la taille de la structure et la rend inférieure à $tsize[i']$. L'identifiant i' doit alors être placé dans L, $tsize[i']$ mis à jour, puis l'étape 2 de l'algorithme, réexécutée.

   En fait, au lieu de réexécuter l'algorithme dès que le tableau *tsize* est modifié, on peut se contenter de maintenir son invariant en mettant L à jour comme expliqué ci-dessus. On peut n'exécuter l'étape 2 de l'algorithme que lorsqu'on désire vraiment disposer des tailles minimales des termes.

4. Nous avons présenté notre méthode de détermination des tailles minimum des termes sur base d'une définition particulière de la taille des termes. D'*autres notions de taille* peuvent être utilisées sans changement notable à apporter à la méthode, à condition que cette taille puisse être déterminée à partir d'une information limitée sur les sous-termes, "propageable incrémentalement". Par exemple, si les termes correspondent à des expressions incluant des opérateurs de diverses priorités, des parenthèses et des noms de fonctions de diverses longueurs, on peut vouloir minimiser le nombre de caractères nécessaires pour écrire l'expression. On peut adapter notre méthode à ce cas de figure en ajoutant d'autres informations au tableau *tsize*, telle que le type des sous-expressions (à définir, selon le contexte).

   Nous avons aussi réalisé d'autres algorithmes utilisant des notions de taille "non compositionnelles" telles que le nombre de nœuds nécessaires pour une représentation partagée des termes (dans laquelle chaque sous-terme n'est représenté qu'une fois). Ces algorithmes sont considérablement plus compliqués et surtout beaucoup moins efficaces. Nous ne les présentons pas ici.

3.1.2 Calcul proprement dit des termes de taille minimale

3.1.2.1 Deux algorithmes simples

Une fois déterminée la fonction size qui donne la taille minimale des termes de chaque ensemble de termes T_i de la dénotation d'une collection de structures E, il est facile, pour chaque identifiant i, de *construire* un terme minimal $t \in T_i$. Un algorithme simple est le suivant.

1. Si $i = i_{null}$, renvoyer *null*.
2. Sinon, choisir une structure $f(i_1, i_2) : i$ telle que
$$\text{relSize}\,(f(i_1, i_2), tsize) = tsize[i].$$
3. Calculer récursivement deux termes minimaux t_1 et t_2 correspondant aux identifiants i_1 et i_2.
4. Renvoyer le terme $f(t_1, t_2)$.

Si l'on veut *écrire* les termes sous forme abrégée, on peut modifier l'algorithme comme suit. Il est à noter qu'on a, dans ce cas, $i \neq i_{null}$, puisque *null* est considéré comme un terme "fictif".

1. Choisir une structure $f(i_1, i_2) : i$ telle que
$$\text{relSize}\,(f(i_1, i_2), tsize) = tsize[i].$$
2. Écrire le symbole f.
3. Si $i_1 \neq i_{null}$, exécuter ce qui suit.
   (a) Écrire une parenthèse ouvrante "(".
   (b) Écrire récursivement le terme minimal t_1 correspondant à l'identifiant i_1.
   (c) Si $i_2 \neq i_{null}$, écrire une virgule ", "; puis, écrire récursivement le terme minimal t_2 correspondant à l'identifiant i_2.
   (d) Écrire une parenthèse fermante ")".

Un terme (minimal) correspondant à structure de la forme $a(i_{null}, i_{null}) : i$ sera écrit sous la forme a, un terme correspondant à une structure de la forme $h(i_1, i_{null}) : i$, telle que $i_1 \neq i_{null}$, sera écrit sous la forme $h(t_1)$; enfin, un terme correspondant à une structure de la forme $f(i_1, i_2) : i$, telle que $i_1, i_2 \neq i_{null}$, sera écrit sous la forme $f(t_1,\ t_2)$.

3.1.2.2 Discussion

Les algorithmes simples proposés à la section précédente peuvent être généralisés pour construire ou écrire toutes les expressions minimales correspondant à un identifiant donné. Ils peuvent aussi être modifiés et étendus pour écrire les termes selon une syntaxe spécifique tenant compte, par exemple, de règles de priorité d'opérateurs. Nous ne donnons pas plus de détails à ce sujet, ici.

3.2 Minimisation des expressions par rapport à une théorie finiment représentable par une collection de structures

Le problème général de la minimisation des expressions, c'est-à-dire de déterminer une expression de taille minimale égale à une expression donnée, dans le cadre d'une théorie précisée, est insoluble efficacement [MS72]. Néanmoins, nous proposons, dans cette section, un cadre limité dans lequel c'est faisable. (Ce cadre peut être élargi à condition d'abandonner l'objectif de minimisation exacte. C'est le sujet de la section suivante.)

Nous modélisons le problème comme suit. Soit *expr*, une expression à simplifier ; par exemple, une expression algébrique, une formule booléenne, ou une expression régulière. Nous pouvons nous efforcer de modéliser la connaissance que nous avons du domaine "logico-mathématique" dans lequel cette expression a un sens, par un ensemble d'égalités entre expressions (équations). Pour être généralement applicables, ces équations doivent contenir des variables, instanciables à des expressions quelconques. Si l'expression *expr* contient des variables, celles-ci jouent en fait le rôle de constantes "non interprétées". Si on ajoute ces constantes aux symboles de fonction et aux constantes des équations, on définit en fait une congruence entre les termes constructibles à partir de ces symboles de fonctions et de ces constantes. Si nous pouvons représenter cette congruence par une collection de structures, il nous est possible de minimiser l'expression *expr* en recherchant une expression minimale qui lui est équivalente, dans la collection de structures.

Dans cette section, nous montrons comment construire cette collection de structures dans le cas où le nombre de classes d'équivalence de la congruence définie par les équations et l'expression à simplifier, est finie. À titre d'exemple, nous appliquons notre méthode au problème de la minimisation des formules booléennes. Nous montrons aussi les limites de notre méthode, dans ce cas : la taille de la collection de structures étant doublement exponentielle sur le nombre de lettres (ou variables propositionnelles) utilisé par la formule booléenne, seules les formules comportant peu de lettres peuvent être minimisées.

3.2.1 Définitions préliminaires

Pour représenter des théories équationnelles, nous avons besoin de termes contenant des variables. Nous complétons donc les définitions données précédemment dans les sections 2.2 et 2.3.

Rappelons que nous notons $\mathcal{T}$, l'ensemble de tous les termes clos. Nous considérons maintenant aussi des termes contenant des variables (et les mêmes constantes et symboles de fonction que précédemment) mais nous n'avons pas besoin de considérer l'ensemble de tous ces termes "comme un tout" et nous ne le notons pas par un symbole précis. Nous notons les variables au moyen des lettres x, y, z, ..., comme d'habitude. Nous n'aurons pas besoin de l'ensemble de toutes les variables. Nous notons $var(t)$, l'ensemble des variables contenues dans le terme t et $var(eq)$, l'ensemble des variables figurant dans l'équation eq.

Définition 28 (Substitutions (closes))**.** Nous appelons *substitution* toute fonction totale σ d'un ensemble de variables V dans un ensemble de termes clos T. Nous notons $V \longrightarrow T$, l'ensemble de toutes ces substitutions (pour V et T, fixés).

L'application d'une substitution σ à un terme t (normalement, mais pas nécessairement, non clos) consiste à remplacer chaque occurrence d'une même variable x par le même terme $\sigma(x)$ dans t. On peut définir cette opération par induction sur la structure des termes, comme suit.

Définition 29 (Application d'une substitution à un terme)**.** Soient $\sigma \in V \longrightarrow T$, une substitution et t, un terme tel que $var(t) \subseteq V$. L'application de σ à t, notée $\sigma(t)$, est définie comme suit.

1. Si t est une variable x, $\sigma(t)$ est simplement $\sigma(x)$.
2. Sinon, t s'écrit sous la forme $f(t_1, t_2)$. Dans ce cas, $\sigma(t)$ est le terme $f(t'_1, t'_2)$ où $t'_k = \sigma(t_k)$ $(k = 1, 2)$.

3.2.2 Congruence définie par une présentation formelle

Nous définissons maintenant la théorie représentée par un ensemble fini d'équations et un ensemble fini de termes clos. Strictement parlant, cette théorie est une congruence, au sens de la définition 8.

Définition 30 (Présentation formelle (d'une congruence))**.** Nous appelons *présentation formelle*, un couple $\langle Eq, C\rangle$, où Eq est un ensemble fini d'équations entre termes non nécessairement clos et C est un ensemble fini de termes clos, complet pour les sous-termes.

Dans la définition ci-dessous, nous notons $ST(Eq)$, l'ensemble de tous les sous-termes des termes figurant dans Eq.

Définition 31 (Congruence définie par une présentation formelle)**.** Soit $\langle Eq, C\rangle$, une présentation formelle. La congruence définie par $\langle Eq, C\rangle$ est

la plus petite congruence $\sim$ telle que l'on ait, pour tous termes s et t et toute substitution σ :

$$(1) \qquad t \in C \implies t \in dom(\sim),$$

$$(2) \qquad \left.\begin{array}{c} t \in ST(Eq) \\ \sigma \in var(t) \longrightarrow dom(\sim) \end{array}\right\} \implies \sigma(t) \in dom(\sim),$$

$$(3) \qquad \left.\begin{array}{c} (s = t) \in Eq \\ \sigma \in var(s = t) \longrightarrow dom(\sim) \end{array}\right\} \implies \sigma(s) \sim \sigma(t).$$

Notons $T = dom(\sim)$. La condition (1) de la définition signifie que les termes de C sont choisis comme "générateurs" pour la construction de T. La condition (3) signifie que l'on obtient des termes équivalents si on instancie les deux termes d'une équation avec les mêmes termes de T. La condition (2) est utile pour assurer que T soit complet pour les sous-termes. Il n'est pas nécessaire d'ajouter une condition du genre

$$\left.\begin{array}{c} t \in ST(Eq) \\ \forall x \in var(t) \ : \ \sigma(x) \sim \sigma'(x) \end{array}\right\} \implies \sigma(t) \sim \sigma'(t),$$

car cette condition est une conséquence du fait que $\sim$ est une congruence.

Note 14. La définition 30 est inspirée de la notion de présentation finie, donnée dans [Wec92], page 143. Nous ajoutons l'adjectif "formelle" pour utiliser un autre terme que dans la définition 13. Les éléments de C sont appelés des *générateurs* dans [Wec92]. Dans les applications que nous avons en vue dans cette section, C est un ensemble fini de *constantes* ne figurant pas dans les équations de Eq. Ce sont donc des constantes "non interprétées". Les paires de termes $\langle s, t\rangle$ telles que $s \sim t$ peuvent être interprétées comme des équations valides dans tous les modèles des équations Eq, les constantes de C y jouant le rôle de variables universellement quantifiées. Si on ne se soucie pas de cette manière de voir les choses, on peut toujours se passer de l'ensemble C (le choisir vide), en rajoutant, dans Eq, des équations de la forme $c = c$ où c est une constante.

Nous allons montrer que la congruence $\sim$ définie par une présentation formelle $\langle Eq, C\rangle$ est égale à la limite (l'union) d'une suite de congruences plus petites pouvant chacune être définie à partir de la précédente, en résolvant un ensemble fini d'équations entre termes clos (sauf, évidemment, la première de ces congruences, définie à partir de C). Il s'en suit que toute ces congruences intermédiaires sont représentables par des collections de structures et que ces collections sont effectivement calculables au moyen des algorithmes que nous avons présentés dans le chapitre 2.

Définition 32 (Complétion partielle d'une congruence par rapport à un ensemble d'équations)**.** Soient $\sim$, une congruence, et Eq, un ensemble fini d'équations entre termes non nécessairement clos. Nous appelons *complétion partielle* de $\sim$ par rapport à Eq, la plus petite congruence $\sim'$ telle que l'on ait, pour tous termes s et t et toute substitution σ :

$$(1) \qquad s \sim t \implies s \sim' t$$

$$(2) \qquad \left.\begin{array}{c} t \in ST(Eq) \\ \sigma \in var(t) \longrightarrow dom(\sim) \end{array}\right\} \implies \sigma(t) \in dom(\sim'),$$

$$(3) \qquad \left.\begin{array}{c} (s = t) \in Eq \\ \sigma \in var(s = t) \longrightarrow dom(\sim) \end{array}\right\} \implies \sigma(s) \sim' \sigma(t).$$

Théorème 25. *Soient $\langle Eq, C\rangle$, une présentation formelle, et $\sim$, la congruence définie par $\langle Eq, C\rangle$. Soit $\sim_0$, la congruence sur C telle que $c \sim_0 c'$ si et seulement si $c = c'$. Soit, enfin, pour chaque entier $i \geq 0$, $\sim_{i+1}$, la complétion partielle de $\sim_i$, par rapport à Eq. On a :*

$$\sim \; = \; \bigcup_{i \geq 0} \sim_i .$$

Preuve du théorème 25. Il est d'abord évident, par récurrence, en confrontant les différentes définitions, que $\sim_i \subseteq \sim$, pour tout $i \geq 0$. Il vient donc qu'on a :

$$\bigcup_{i \geq 0} \sim_i \; \subseteq \; \sim .$$

On peut ensuite montrer facilement que la relation

$$\bigcup_{i \geq 0} \sim_i$$

est une congruence et qu'elle vérifie les trois conditions de la définition de $\sim$, sauf peut-être la minimalité. (Nous laissons les "détails" au lecteur.) On a donc, aussi :

$$\sim \; \subseteq \; \bigcup_{i \geq 0} \sim_i .$$

□

Théorème 26. *Soit $\sim$, une congruence possédant un nombre fini de classes d'équivalence, utilisant un nombre fini de symboles de fonction, et dont le domaine est complet pour les sous-termes. Cette congruence est représentable par une collection de structures bien formée et normalisée E. Plus précisément, elle est égale à la dénotation abstraite de cette collection de structures E.*

Preuve du théorème 26. On construit une collection de structures représentant $\sim$, comme suit.

1. À chaque classe d'équivalence Cl de $\sim$, on associe un identifiant d'ensembles de structures distinct i_{Cl}. Soit I, l'ensemble de ces identifiants.
2. À chaque terme $f(t_1, t_2)$, appartenant au domaine de $\sim$, on associe la structure $f(i_{Cl_1}, i_{Cl_2}) : i_{Cl}$ où $t_k \in Cl_k$ $(k = 1, 2)$. (Cette structure est correctement définie parce que le domaine de $\sim$ est complet pour les sous-termes.)

L'ensemble d'identifiants I, défini au point 1, est fini puisque le nombre de classes d'équivalences de $\sim$ l'est. L'ensemble des structures définies au point 2 est également fini puisque l'ensemble des identifiants du point 1 est fini ainsi que celui des symboles de fonction. Donc, ces structures constituent une collection de structures E utilisant l'ensemble d'identifiants I.

Notons Cl_i la classe d'équivalence de $\sim$ qui est telle que $i = i_{Cl_i}$. Nous prouvons que $(Cl_i)_{i \in I}$ est égale à la dénotation concrète de E. Pour le montrer, il suffit de prouver que pour n'importe quel terme t et n'importe quel identifiant $i \in I$, on a : $t \in Cl_i$ si et seulement si il existe une structure $f(i_1, i_2) : i$, appartenant à E telle que $t = f(t_1, t_2)$ et $t_j \in Cl_{i_j}$ $(j = 1, 2)$. Si $f(t_1, t_2) \in Cl_i$, il existe i_1, i_2 tels que $t_j \in Cl_{i_j}$ $(j = 1, 2)$, parce que T est complet pour les sous-termes. De plus, E contient la structure $f(i_1, i_2) : i$. Dans l'autre sens, si $t_j \in Cl_{i_j}$ $(j = 1, 2)$ et $f(i_1, i_2) : i \in E$, il existe des termes s_1 et s_2 tels que $s_j \in Cl_{i_j}$ $(j = 1, 2)$ et $f(s_1, s_2) \in Cl_i$, par définition de E. On a évidemment, $s_j \sim t_j$ $(j = 1, 2)$. Donc, on a aussi : $f(s_1, s_2) \sim f(t_1, t_2)$, puisque $\sim$ est une congruence. D'où, finalement, $f(t_1, t_2) \in Cl_i$.

La collection de structures E est normalisée car si elle contenait deux structures distinctes de même clé, la relation $\sim$ contiendrait deux classes d'équivalence distinctes contenant des termes équivalents, ce qui est impossible. □

Lemme 12 (Calcul de la complétion partielle d'une congruence)**.** *Soient E, une collection de structures normalisée et $\sim$, sa dénotation abstraite. Soit aussi Eq, un ensemble fini d'équations entre termes non nécessairement clos. On peut calculer effectivement une collection de structures E' dont la dénotation abstraite $\sim'$ est égale à la complétion partielle de $\sim$, par rapport à Eq, en résolvant un nombre* fini *d'équations entre termes clos, par rapport à E.*

Preuve du lemme 12. Soit $(T_i)_{i \in I}$, la dénotation concrète de E. Nous

choisissons un terme t_i, dans chaque ensemble T_i et nous notons $\hat{T}$, l'ensemble de ces termes. L'ensemble $\hat{T}$ est fini. Par conséquent, l'ensemble des substitutions σ telles que $\sigma \in var(eq) \longrightarrow \hat{T}$ est également fini. Finalement, l'ensemble des équations de la forme $\sigma(s) = \sigma(t)$ où $s = t$ est une équation de Eq et $\sigma \in var(eq) \longrightarrow \hat{T}$ est fini.

Il est donc possible de résoudre en un temps fini l'ensemble d'équations défini ci-dessus en appliquant l'algorithme de la section 2.3.3. La résolution de ces équations crée une collection de structures E'' dont la dénotation abstraite $\sim''$ contient $\sim$ et est telle que $\sigma(s) \sim'' \sigma(t)$ pour toute équation $s = t \in Eq$ et toute substitution $\sigma \in var(eq) \longrightarrow \hat{T}$. C'est la plus petite congruence vérifiant ces propriétés. On a donc $\sim'' \subseteq \sim'$ où $\sim'$ est égale à la complétion partielle de $\sim$, par rapport à Eq.

Nous prouvons maintenant que $\sim' \subseteq \sim''$ en montrant que toute preuve d'une affirmation de la forme $u \sim' v$, utilisant les conditions de la définition 32 peut être transformée en une preuve de $u \sim'' v$. (Nous utilisons ici la notion de preuve correspondant à la définition 10.) La seule différence entre les deux sortes de preuves est que là où une preuve de $u \sim' v$ peut utiliser une affirmation intermédiaire de la forme $\sigma(s) \sim' \sigma(t)$ avec $s = t \in Eq$ et $\sigma \in var(s = t) \longrightarrow dom(\sim)$, une preuve de $u \sim'' v$ peut seulement utiliser une affirmation de la forme $\sigma'(s) \sim'' \sigma'(t)$ avec $\sigma' \in var(s = t) \longrightarrow \hat{T}$. On peut néanmoins en déduire qu'on a aussi $\sigma(s) \sim'' \sigma(t)$ car à toute substitution $\sigma \in var(s = t) \longrightarrow dom(\sim)$, on peut faire correspondre une substitution $\sigma' \in var(s = t) \longrightarrow \hat{T}$ telle que $\sigma'(x) \sim \sigma(x)$, pour toute variable $x \in var(s = t)$, en choisissant pour $\sigma'(x)$ le terme de $\hat{T}$ équivalent à $\sigma(x)$, pour la relation $\sim$. Dès lors, $\sigma(s) \sim'' \sigma'(s) \sim'' \sigma'(t) \sim'' \sigma(t)$. (Les deux équivalences $\sigma(s) \sim'' \sigma'(s)$ et $\sigma'(t) \sim'' \sigma(t)$ se prouvent par induction sur la structure des termes, en utilisant les deux faits que $\sim \subseteq \sim''$ et que $\sim''$ est une congruence.) □

Il résulte du lemme 12 qu'il est possible de calculer effectivement des collections de structures $E_0, E_1, \ldots, E_i, \ldots$ représentant (dénotant) les congruences $\sim_0, \sim_1, \ldots, \sim_i, \ldots$ définies dans l'énoncé du théorème 25. Nous allons donner un algorithme pour le faire qui est beaucoup plus efficace que la méthode suggérée dans le lemme. Mais avant, nous utilisons le lemme pour démontrer que la congruence définie par une présentation formelle $\langle Eq, C \rangle$ est effectivement calculable par une collection de structures si elle comporte un nombre fini de classes d'équivalence.

Théorème 27. *Sous les mêmes hypothèses que celles du théorème 25, si le nombre de classes d'équivalence de la congruence $\sim$ est fini, il existe un entier j tel que $\sim_j = \sim$.*

Il s'en suit que la congruence $\sim$ peut être calculée effectivement sous

forme d'une collection de structures.

Preuve du théorème 27. Supposons que la congruence $\sim$ possède un nombre fini de classes d'équivalences. Posons $T = dom(\sim)$ et $T^j = dom(\sim_j)$ pour tout $j \geq 0$. On a évidemment $T^0 \subseteq T^1 \subseteq \ldots \subseteq T^j \subseteq \ldots T$.

Pour chaque classe d'équivalence Cl de $\sim$, nous pouvons choisir un terme $t_{Cl} \in Cl$. Ensuite, pour n'importe quel terme $f(t_1, t_2)$, appartenant à T, nous pouvons choisir le terme $f(t_{Cl_1}, t_{Cl_2})$ tel que $t_k \in Cl_k$ $(k = 1, 2)$, qui appartient aussi à T puisque $\sim$ est une congruence. Comme le nombre de classes d'équivalence de $\sim$ est fini, tous ces termes sont en nombre fini. Chaque terme $f(t_{Cl_1}, t_{Cl_2})$ appartient à une certaine classe d'équivalence Cl. Donc, il existe, pour chaque terme de cette sorte un j tel que $f(t_{Cl_1}, t_{Cl_2}) \sim_j t_{Cl}$. Comme le nombre de paires de termes $f(t_{Cl_1}, t_{Cl_2})$, t_{Cl} est fini, il existe un j suffisamment grand pour avoir $f(t_{Cl_1}, t_{Cl_2}) \sim_j t_{Cl}$ pour toutes ces paires de termes. Fixons un tel j.

La congruence $\sim_j$ est représentable par une collection de structures E^j, effectivement calculable. Soit $(T_i^j)_{i \in I^j}$, sa dénotation concrète. Notons i_{Cl}, l'identifiant tel que $t_{Cl} \in T^j_{i_{Cl}}$. La collection de structures E^j contient nécessairement toutes les structures de la forme $f(i_{Cl_1}, i_{Cl_2}) : i_{Cl}$. La sous-collection formée de ces structures a pour dénotation la congruence $\sim$ (voir la démonstration du théorème 26). Cela implique que $T \subseteq T^j$. D'où $T^j = T$. Dès lors, la collection de structures E^j ne peut contenir d'autres structures que les structures formant la sous-collection et elle a pour dénotation la congruence $\sim$. De plus, elle est effectivement (finiment) calculable. □

3.2.3 Calcul efficace de la congruence définie par une présentation formelle

3.2.3.1 Théorie du problème

Soit $\sim$, la congruence définie par la présentation formelle $\langle Eq, C \rangle$. Si cette congruence possède un nombre fini de classes d'équivalences, nous pouvons la calculer plus efficacement qu'en construisant et en résolvant des équations entre termes clos, comme expliqué plus haut. Au lieu de choisir des termes dans les ensembles de termes représentés par les identifiants de la collection de structures courante, nous pouvons utiliser directement les identifiants eux-mêmes, sans construire de termes explicitement. Pour mettre cela en forme, nous introduisons la notion de valuation et nous généralisons l'opération *toSet* pour qu'elle s'applique à un terme non clos et à une valuation.

Définition 33 (Valuations)**.** Nous appelons *valuation* toute fonction totale *val* d'un ensemble de variables V dans un ensemble d'identifiants d'ensembles de structures I. Nous notons $V \longrightarrow I$, l'ensemble de toutes ces valuations (pour V et I, fixés).

Soit E, une collection de structures utilisant l'ensemble d'identifiants I. Soit $(T_i)_{i\in I}$, la dénotation concrète de E et soit $T = \cup_{i\in I} T_i$. Toute valuation $val \in (V \longrightarrow I)$ peut alors être vue comme représentant l'ensemble de toutes les (ou une quelconque des) substitutions $\sigma \in (V \longrightarrow T)$ telles que $\sigma(x) \in T_{val(x)}\ \forall x \in V$.

Définition 34 (Opération *toSet* applicable à un terme et une valuation)**.** Soit E, une collection de structures normalisée utilisant l'ensemble d'identifiants I. Soient aussi t, un terme non nécessairement clos, et $val \in (var(t) \longrightarrow I)$, une valuation. L'opération *toSet*, appliquée à E, t et *val*, modifie éventuellement la collection E (en lui rajoutant de nouvelles structures, uniquement) et renvoie un identifiant. Elle se calcule récursivement comme suit :

1. Si t est une variable x, la collection E n'est pas modifiée et on renvoie l'identifiant $val(x)$.
2. Sinon, t est de la forme $f(t_1, t_2)$ [1]. On applique récursivement l'opération *toSet* à E, t_j et *val* (pour $j = 1, 2$). Soient i_1 et i_2, les identifiants renvoyés. Si la collection de structures (éventuellement modifiée, entretemps) contient une structure de la forme $f(i_1, i_2) : i$, on renvoie l'identifiant i. Sinon, on choisit un nouvel identifiant i (non encore utilisé), on ajoute la structure $f(i_1, i_2) : i$ à la collection de structures et on renvoie i.

Théorème 28 (Spécification de l'opération *toSet*, généralisée)**.** *Reprenons les notations de la définition 34, et soient $(T_i)_{i\in I}$ la dénotation concrète de E (initialement) et $T = \cup_{i\in I} T_i$. L'application de l'opération toSet à E, t et val, équivaut exactement à appliquer l'opération toSet simple (voir la définition 22) au terme $\sigma(t)$ où la substitution $\sigma \in (var(t) \longrightarrow T)$ est telle que $\sigma(x) \in T_{val(x)}\ \forall x \in var(t)$.*

Preuve. La preuve est immédiate, par induction sur la structure du terme t. □

Définition 35 (Résolution d'une équation par rapport à une valuation)**.** Soient $s = t$, une équation entre termes non nécessairement clos, E, une

1. Rappelons que nous avons choisi de ne pas traiter explicitement le cas du terme spécial *null*, dans les définitions et les preuves, laissant cet exercice "au lecteur". (Dans ce cas-ci, l'opération renvoie i_{null} lorsque $t = null$.)

collection de structures normalisée, utilisant l'ensemble d'identifiants I et $val \in (var(s = t) \longrightarrow I)$, une valuation. L'opération de *résolution de l'équation $s = t$, par rapport à E et val* s'exécute comme suit :

1. On applique l'opération *toSet* à E, s et val, puis à E (éventuellement modifiée), t et val.
2. Soient i_s et i_t, les identifiants renvoyés par les deux applications de l'opération *toSet*, ci-dessus. On applique l'opération *unify*[2] à E (résultant de l'exécution du point 1), et aux identifiants i_s et i_t.

Théorème 29. *Avec les notations de la définition 35 et du théorème 28, l'effet de la résolution de l'équation $s = t$, par rapport à E et val modifie la collection de structures E exactement de la même façon que la résolution de l'équation entre termes clos*[3] $\sigma(s) = \sigma(t)$ *où la substitution $\sigma \in (var(s = t) \longrightarrow T)$ est telle que $\sigma(x) \in T_{val(x)}$ $\forall x \in var(s = t)$.*

Preuve. Conséquence immédiate des définitions et du théorème 28. □

3.2.3.2 Algorithme de calcul de la complétion partielle

Nous donnons maintenant un algorithme pour calculer la complétion partielle d'une congruence représentée par une collection de structures normalisée E par rapport à un ensemble fini d'équations entre termes non nécessairement clos Eq.

Nous supposons avoir défini un ordre total sur l'ensemble des variables figurant dans les équations de Eq. Nous notons x_i, la i-ème variable selon cet ordre (avec i, un entier strictement positif). Nous supposons aussi que si une équation utilise exactement k variables, ce sont les variables $x_1, x_2, \ldots, x_k$.

L'algorithme reçoit au départ, outre l'ensemble d'équations Eq, une collection de structures normalisée E, utilisant un ensemble d'identifiants I. Il modifie progressivement la collection de structures en résolvant des équations par rapport à des valuations. De nouveaux identifiants sont ajoutés à la collection par des exécutions de l'opération *toSet*, d'autres sont retirés de la collection par des exécutions de l'opération *unify*. Nous supposons que lorsque l'opération *toSet* ajoute une nouvelle structure $f(i_1, i_2) : i$, à la collection, l'identifiant choisi i est toujours "nouveau". On ne réutilise donc pas les identifiants figurant dans I au départ. À un moment de l'exécution de l'algorithme, nous notons I l'ensemble courant des identifiants de la collection de structures actuelle. Par contre,

2. Voir section 2.3.2.4.
3. Voir section 2.3.3.

nous notons J, l'ensemble des identifiants de I qui y figuraient déjà, au départ. Nous construisons des valuations à partir des éléments de J, uniquement. Le choix d'une nouvelle valuation se fait comme suit : on choisit d'abord un entier k tel qu'au moins une équation de Eq contient k variables exactement ; on choisit ensuite k identifiants, non nécessairement distincts, i_1, i_2, ..., i_k, dans J. La valuation choisie est telle que $val(x_j) = i_j$ $(j = 1, 2 \ldots, k)$. Enfin, on suppose qu'on ne choisit jamais deux fois la même valuation. (Nous laissons indéfinie ici la manière exacte de s'y prendre, mais nous en discutons plus loin.) Avec tous ces présupposés, l'algorithme est simplement le suivant.

Tant qu'il est possible de choisir une nouvelle valuation, on fait ce qui suit.

1. Choisir une valuation val.
2. Choisir, dans Eq, une équation $s = t$, telle que $var(s = t)$ est égal au domaine de val, non déjà choisie pour cette valuation.
3. Résoudre l'équation $s = t$, par rapport à E et val.
4. S'il reste au moins une équation, telle que spécifiée au point 2, à choisir, et si tous les identifiants figurant dans le codomaine de val appartiennent toujours à J, reprendre au point 2. Sinon, il ne reste rien à faire pour cette valuation.

Nous prouvons maintenant la correction de l'algorithme.

Théorème 30. *Soient Eq, un ensemble fini d'équations non nécessairement closes, E, une collection de structures normalisée, et $\sim$, la dénotation abstraite de E. L'exécution de l'algorithme présenté ci-dessus, appliqué à E et Eq, modifie la collection de telle sorte qu'elle dénote abstraitement la complétion partielle de $\sim$, par rapport à Eq (voir la définition 32).*

Preuve du théorème 30.

1. L'algorithme se termine car le nombre d'équations et le nombre de valuations qu'il est possible de choisir sont finis.
2. Notons I_0 l'ensemble d'identifiants utilisés par la collection de structures initiale. Et notons $(T_i^0)_{i \in I_0}$ sa dénotation concrète initiale. Choisissons un terme t_i dans chaque ensemble T_i^0 et posons $\hat{T} = \{t_i \mid i \in I_0\}$. À chaque valuation $val \in (var(eq) \longrightarrow I_0)$, où $eq \in Eq$, nous pouvons associer la substitution $\sigma_{val} \in (var(eq) \longrightarrow \hat{T})$, en posant $\sigma(x) = t_{val(x)}$ $\forall x \in var(eq)$. Pour calculer la complétion partielle de $\sim$ par rapport à Eq, il suffit de résoudre, par rapport à E (dans son état initial), toutes les équations entre termes clos, de la forme $\sigma_{val}(s) = \sigma_{val}(t)$, pour toutes les équations

$s = t \in Eq$ et toutes les valuations $val \in (var(eq) \longrightarrow I_0)$ (voir le lemme 12).

3. L'algorithme que nous avons donné résout les équations de Eq par rapport à un sous-ensemble des valuations $val \in (var(eq) \longrightarrow I_0)$. (Et non pas par rapport à toutes, car, lorsque l'ensemble d'identifiants J est amputé de certains identifiants, suite à une exécution de l'opération *unify*, les valuations qui utilisaient ces identifiants ne sont plus applicables.) Chacune de ces résolutions d'une équation non close $s = t$, par rapport à une valuation val, équivaut exactement à résoudre l'équation entre termes clos $\sigma_{val}(s) = \sigma_{val}(t)$ (voir le théorème 29).
4. Il reste à montrer que les résolutions d'équations qui n'ont pas été effectuées ne sont pas nécessaires, c'est-à-dire qu'elles ne modifieraient pas la collection de structures obtenue à la fin de l'algorithme. Cela revient à montrer que les résolutions d'équations closes qui leur correspondent ne le font pas. Soit $\sigma_{val}(s) = \sigma_{val}(t)$, une telle équation. On peut associer à la valuation val, une valuation $val' \in (var(s = t) \longrightarrow J)$, où J représente l'ensemble des identifiants de I_0 toujours utilisés par la collection à la fin de l'algorithme, en choisissant pour $val'(x_i)$, l'identifiant j_i tel que les termes $t_{val(i)}$ et t_{j_i} (de l'ensemble $\hat{T}$) sont équivalents par rapport à la dénotation finale de la collection (i.e., l'identifiant $val(x_i)$ a été renommé en j_i). Les termes $\sigma_{val}(s)$ et $\sigma_{val'}(s)$ sont donc équivalents par rapport à la dénotation finale de la collection. De même pour les termes $\sigma_{val}(t)$ et $\sigma_{val'}(t)$. Comme tous les identifiants utilisés par val' sont toujours présents dans la collection à la fin de l'algorithme, l'équation $s = t$ a été résolue par rapport à val' (sinon l'exécution de l'algorithme ne serait pas encore terminée). Il s'en suit que les termes $\sigma_{val'}(s)$ et $\sigma_{val'}(t)$ sont équivalents à la fin de l'algorithme. Donc aussi les termes $\sigma_{val}(s)$ et $\sigma_{val}(t)$. Par conséquent, l'équation $\sigma_{val}(s) = \sigma_{val}(t)$ est "trivialement vérifiée" dans la collection de structures finale et sa résolution ne modifierait donc pas celle-ci.
5. Le raisonnement fait au point 4, ci-dessus suppose que lorsqu'une opération *substitute* est appliquée à deux identifiants k et l, durant l'exécution de l'algorithme, et qu'au moins un de ces deux identifiants appartient à J, l'opération choisit de conserver un identifiant appartenant à J, sans quoi certaines valuations $val \in (V \longrightarrow J)$ pourraient devenir inapplicables sans être remplacées par une valuation équivalente : il faut que chaque identifiant $val(x_i)$ ait été fusionné à un identifiant appartenant toujours à J. Il est possible

de satisfaire cette exigence en choisissant toujours de conserver l'identifiant le plus ancien (celui qui a été introduit le premier dans l'ensemble des identifiants de la collection). (Voir la remarque 2, ci-dessous.) □

Nous donnons maintenant des informations complémentaires sur la façon concrète d'implémenter l'algorithme "de haut niveau" donné ci-dessus.

1. Il est souhaitable de générer efficacement les valuations utilisées par l'algorithme et aussi d'en générer le moins possible. De plus, comme on l'a montré expérimentalement dans la section 2.5.6, il est généralement préférable de résoudre les "plus petites équations" d'abord. Il semble dès lors raisonnable d'utiliser la méthode suivante. Nous construisons, d'abord, une liste formée des identifiants de la collection initiale, triée en ordre croissant des termes de taille minimale correspondant à ces identifiants. Nous parcourons ensuite cette liste élément par élément et pour chaque élément, nous générons toutes les valuations utilisant au moins une fois cet élément (cet identifiant) et uniquement des éléments qui le précèdent dans la liste. Nous pouvons le faire avec une instance de la classe `OneList` (voir la section 2.4.4), ce qui permet de supprimer de la liste les identifiants enlevés de la collection sans perturber le parcours de la liste.

   En procédant dans cet ordre, nous pouvons, de plus, diminuer le nombre de valuations à considérer en modifiant le critère de choix de l'identifiant à remplacer par l'autre, lors d'une application de l'opération *substitute* à deux identifiants k et l (voir la section 2.5.1 et, en particulier, la note à la fin de cette section). Au lieu de conserver l'identifiant qui a le plus d'occurrences dans la collection de structures courante, nous choisissons celui qui vient en tête dans la liste des identifiants. Cela permet de supprimer de la liste certains identifiants sans jamais avoir eu à générer de valuations qui les utilisent. Le choix inverse conduit à résoudre plusieurs fois les mêmes équations avec des valuations équivalentes. (Les résolutions différentes de la première sont sans effet sur la collection. Elles sont donc peu coûteuses mais néanmoins superflues. Et elles peuvent être très nombreuses.) Nous verrons expérimentalement que cette façon de faire peut donner de meilleurs résultats que la méthode préconisée dans la section 2.5.1. Toutefois, les deux méthodes ne sont pas "si différentes" en pratique puisque les identifiants figurant en tête de liste vont rapidement voir leur nombre d'occurrences augmenter.

2. Nous avons supposé, dans les présupposés de l'algorithme, que l'opération *toSet* choisit toujours un nouvel identifiant i lors de la création d'une nouvelle structure $f(i_1, i_2) : i$. Or cela est contradictoire avec nos explications données dans la section 2.4.4, où il est dit que les identifiants "libérés" lors de l'exécution d'une opération *substitute* sont placés dans une liste d'identifiants libres pour être réutilisés. Cette possibilité est, en fait, essentielle pour permettre le traitement d'un très grand nombre d'équations. Pour résoudre ce dilemme, nous ajoutons aux structures de données "concrètes", décrites à la section 2.4.4, un tableau d'entiers de `NBRSETS` éléments qui fournit la "date" à laquelle un identifiant est "(re-)mis en service" et nous distinguons nouveaux et anciens identifiants sur base de cette date.
3. Il arrive qu'il ne soit pas nécessaire de considérer autant de valuations que demandé dans l'algorithme. Par exemple, si certains symboles de fonction représentent des opérations associatives et commutatives, certaines valuations peuvent être redondantes. Cela peut permettre d'accélérer l'algorithme mais ne peut être justifié que dans des contextes particuliers. Nous en donnons des exemples plus loin.

3.2.3.3 Calcul complet de la congruence

Soit $\langle Eq, C\rangle$, une présentation formelle. L'algorithme ci-dessous se termine si la congruence définie par $\langle Eq, C\rangle$ possède un nombre fini de classes d'équivalence (voir la définition 31).

1. Créer une collection de structures initiale E ayant pour dénotation abstraite la congruence $\sim_0 = \{\langle c, c\rangle \mid c \in C\}$.
2. Appliquer l'algorithme de calcul de la complétion partielle à la collection E et à l'ensemble d'équations Eq.
3. Si la collection de structures n'a pas été modifiée par la dernière exécution du point 2, l'algorithme se termine. Sinon, reprendre l'exécution au point 2.

Théorème 31. *Si la congruence $\sim$ définie par $\langle Eq, C\rangle$ possède un nombre fini de classes d'équivalence, l'algorithme ci-dessus se termine et crée une collection de structures dont la dénotation abstraite est la congruence $\sim$.*

Preuve. Nous utilisons les notations du théorème 25. D'après le théorème 30, la $j-$ème exécution du point 2 de l'algorithme modifie la collection de structures E de telle sorte que sa dénotation abstraite soit

égale à la congruence $\sim_j$. Par ailleurs, d'après le théorème 27, il existe un j tel que $\sim_j = \sim$, si le nombre de classes d'équivalence de $\sim$ est fini. Donc, l'algorithme se termine après $j+1$ itérations où j est le plus petit entier tel que $\sim_j = \sim$. □

Au niveau de l'implémentation concrète, nous prenons en compte les points suivants, pour augmenter l'efficacité de l'algorithme.

1. Lors de chaque exécution du point 2 de l'algorithme, certains identifiants présents au début de l'exécution sont retirés de l'ensemble des identifiants utilisés par la collection par l'opération *unify*, d'autres lui sont rajoutés par l'opération *toSet*. La condition $\sim_i = \sim$ est vérifiée, à la fin d'une exécution du point 2, si et seulement si l'ensemble des nouveaux identifiants est vide. En effet, comme nous l'avons expliqué dans la preuve de correction du théorème 30, toutes les équations de l'ensemble *Eq* ont été résolues par rapport à toutes les valuations qu'on peut construire avec les anciens identifiants. Donc, aucune résolution d'équation ne peut plus modifier la collection de structures si l'ensemble des nouveaux identifiants est vide. Il n'est donc pas nécessaire de réexécuter une fois de plus le point 2, pour s'assurer que $\sim_j = \sim$.
2. Lorsqu'on réexécute le point 2 de l'algorithme, il n'est pas nécessaire de résoudre les équations avec toutes les valuations possibles mais uniquement avec celles qui contiennent au moins un nouvel identifiant (puisque les équations sont trivialement vérifiées pour les autres valuations). Pour organiser ce calcul efficacement, on peut diviser la liste des identifiants qu'on utilise pour construire les valuations, en un préfixe contenant les anciens identifiants et un suffixe contenant les nouveaux. Pour identifier la frontière entre les deux, on utilise la date de mise en service des identifiants et on mémorise la date courante au début de chaque exécution du point 2.

3.2.4 Minimisation des formules booléennes

Dans cette section, nous utilisons les algorithmes présentés dans les sections précédentes pour minimiser les formules booléennes contenant un nombre limité de lettres (ou "variables propositionnelles"). L'équivalence logique entre formules booléennes utilisant un ensemble déterminé et fini de lettres constitue une congruence possédant un nombre fini de classes d'équivalences. Nous pouvons donc, en théorie au moins, lui appliquer nos algorithmes.

Nous commençons par calculer une collection de structures qui représente toutes les formules booléennes que l'on peut construire en utilisant les lettres a, b et c. Nous appliquons donc l'algorithme de la section 3.2.3 à la présentation formelle $\langle Eq, C\rangle$ où $C = \{a, b, c\}$ et l'ensemble d'équations Eq est donné à la figure 3.1. Cet ensemble d'équations peut être vu

$$
\begin{array}{rcl}
x\,0 & = & 0 \\
x + 0 & = & x \\
x + 1 & = & 1 \\
x\,\overline{x} & = & 0 \\
\overline{\overline{x}} & = & x
\end{array}
\qquad\qquad
\begin{array}{rcl}
x\,y & = & y\,x \\
(x\,y)\,z & = & x\,(y\,z) \\
x + y & = & y + x \\
(x + y) + z & = & x + (y + z) \\
\overline{x\,y} & = & \overline{x} + \overline{y} \\
x + y\,z & = & (x + y)\,(x + z)
\end{array}
$$

Figure 3.1 – Un ensemble d'équations pour le calcul booléen

comme un ensemble d'axiomes pour le calcul booléen mais il n'est pas minimal. Le choix de cet ensemble donne de bons résultats en termes de temps d'exécution et d'utilisation de la mémoire, comme nous allons le montrer. Nous analyserons aussi quelques autres possibilités, plus loin.

La table 3.1 donne des mesures expérimentales concernant l'application de notre algorithme au calcul de la congruence définie par la présentation formelle $\langle Eq, C\rangle$, spécifiée ci-dessus. La colonne i indique combien de fois l'algorithme de calcul de la complétion partielle a déjà été exécuté. Les colonnes M et N donnent respectivement le nombre d'identifiants et le nombre de structures utilisées par la collection à ce moment, tandis que les colonnes M_{max} et N_{max} donnent les maximums de ces valeurs durant la i-ème exécution de l'algorithme. La colonne $\sharp\, J$ indique combien d'identifiants figurant dans la collection au début de la i-ème exécution, y figurent toujours à la fin de celle-ci. (Ce sont les identifiants qui n'ont pas été renommés et qui n'ont pas été choisis pour créer de nouvelles structures.) La colonne N_{new} indique combien de nouvelles structures ont été ajoutées, depuis le début, à la collection de structures, par les différentes exécutions de l'opération *toSet*. Finalement, la colonne t donne, en secondes, le temps d'exécution, depuis le début de l'exécution de l'algorithme complet, jusqu'à la fin de la i-ème exécution de l'algorithme de calcul de la complétion partielle. On voit d'abord que le temps d'exécution est raisonnable vu la taille de la collection de structures à calculer (inférieur à 5 secondes). On vérifie ensuite que le résultat est conforme aux attentes. En effet, le nombre de classes d'équivalences de l'ensemble des formules booléennes utilisant 3 lettres est égal à 256, ce qui est bien

i	M	N	M_{max}	N_{max}	N_{new}	$\sharp\, J$	t
0	5	5	5	5	5	0	0,00
1	144	307	145	307	308	5	0,00
2	859	4.642	1.013	5.174	6.804	21	0,29
3	783	110.084	2.201	110.084	216.488	169	2,27
4	281	134.053	935	142.265	339.004	256	4,87
5	256	131.333	282	134.056	339.022	256	4,87

Table 3.1 – Mesures pour les formules booléennes utilisant les lettres a, b et c

i	$\sharp\, eq_{th}$	$\sharp\, eq_{exec}$	$\sharp\, eq_{triv}$	$\sharp\, eq_M^+$	$\sharp\, eq_M^-$	$\sharp\, eq_N^+$	$\sharp\, eq_N^-$	δ_M^+	δ_M^-	δ_N^+	δ_N^-
1	175	175	12	131	3	160	0	2	1	3	0
2	1.556.105	8.480	3.447	1.642	316	4.717	75	2	157	3	539
3	319.133.864	2.495.102	2.293.903	14.330	8.270	192.929	3.167	2	282	3	3.498
4	239.369.128	6.086.841	5.965.634	3.420	3.983	117.224	1.996	2	63	3	1.020
5	2.745.875	30	6	0	6	18	6	0	6	1	558

Table 3.2 – Statistiques relatives à l'exécution des équations

la valeur finale de M. Ensuite, la représentation de l'ensemble des formules booléennes sous forme d'une collection de structures comporte effectivement 131.333 structures car il y a $256 \times 256 = 65.536$ structures de la forme $+(i_1, i_2) : i$, puis 65.536 structures de la forme $.(i_1, i_2) : i$, 256 structures de la forme $!(i_1) : i$, et 1 structure de chacune des formes 0, 1, a, b, c. Cela donne, au total, $65.536 + 65.536 + 256 + 5 = 131.333$ structures. En conséquence, toutes les formules booléennes utilisant les trois lettres a, b, c (au plus) sont représentées dans la collection de structures calculée par l'algorithme. Donc, pour minimiser n'importe laquelle de ces formules, il suffit de lui appliquer l'opération *toSet* et de choisir une formule de taille minimale dans la classe d'équivalence T_i correspondant à l'identifiant i renvoyé par l'opération. Un examen complémentaire de la table 3.1 montre que notre algorithme, appliqué à ce choix particulier d'équations pour le calcul booléen, se comporte de manière relativement "clairvoyante" et économe en mémoire. Il est normal que le nombre d'identifiants augmente d'abord car les classes de formules équivalentes comportent trop peu de formules pour être fusionnées grâce à la présence de formules communes. Toutefois, ce nombre reste raisonnable (inférieur ou égal à 2.201) et on évite "l'explosion combinatoire". Le nombre de structures augmente progressivement sans dépasser notablement le nombre final à atteindre mais suffisamment vite pour favoriser la fusion des classes d'équivalence. L'examen de la colonne $\sharp\, J$ montre

qu'à la fin de la troisième itération ($i = 3$), une grande partie de la collection finale est déjà constituée et que c'est quasi chose faite, à la fin de la quatrième itération : il reste seulement quelques ensembles de structures à intégrer (fusionner) aux 256 déjà créés à la fin de la troisième itération.

Nous analysons maintenant le processus d'exécution des équations par rapport aux valuations. Les mesures réalisées à ce sujet sont rapportées à la table 3.2. La colonne $\sharp\, eq_{th}$ fournit le nombre "théorique" de couples équation-valuation à exécuter, dans l'hypothèse où aucune valuation ne serait rendue inapplicable, en cours de route, suite à la fusion de deux identifiants figurant dans la collection au départ (de l'itération i). La colonne $\sharp\, eq_{exec}$ donne le nombre de couples effectivement exécutés. On voit qu'il y en a énormément moins que de couples "théoriques" : jusqu'à presque 100.000 fois moins à la dernière itération. Cela montre l'intérêt de "travailler directement" sur les identifiants plutôt qu'avec des équations sur des termes clos ou même avec une structure Union-Find mettant en relation les identifiants initiaux et les identifiants courants. La colonne $\sharp\, eq_{triv}$ fournit le nombre d'exécutions de couples équation-valuation qui ne changent pas la collection de structures et qui sont donc inutiles (*a posteriori*). On voit clairement qu'il y a énormément beaucoup de telles exécutions particulièrement dans les phases "d'intense activité" de l'algorithme. Cela suggère que d'autres formes d'optimisation pourraient être mises en œuvre pour détecter quand certaines équations ne sont plus effectives. Les colonnes $\sharp\, eq^+_M$, $\sharp\, eq^-_M$, $\sharp\, eq^+_N$ et $\sharp\, eq^-_N$ donnent, respectivement, le nombre d'exécutions de couples équation-valuation qui augmentent (diminuent) le nombre d'identifiants ($_M$) et le nombre de structures ($_N$) de la collection. Finalement, les colonnes δ^+_M, δ^-_M, δ^+_N et δ^-_N donnent, respectivement, le nombre maximum d'identifiants ($_M$) et de structures ($_N$) qui sont ajoutés ($^+$) ou retirés ($^-$) de la collection par *une seule* exécution d'un couple équation-valuation. On voit que peu d'identifiants et de structures sont rajoutés à la fois, ce qui est normal vu la forme des équations. Il y a aussi relativement peu d'équations qui ajoutent effectivement un nouvel identifiant, ce qui contribue à la convergence rapide de l'algorithme. Par contre, certaines exécutions (rares) retirent un grand nombre d'identifiants et de structures de la collection, grâce à l'opération *normalize*. Ces exécutions correspondent aux "moments-clés" de la mise en forme de la collection de structures.

Nous analysons maintenant trois variantes de l'exemple "de référence" analysé en détails ci-dessus, afin de mettre en lumière certains points délicats dans l'application de notre méthode.

L'ensemble d'équations proposé dans la figure 3.1 ne constitue pas un

i	M	N	M_{max}	N_{max}	N_{new}	$\sharp\, J$	t
0	5	5	5	5	5	0	0,0
1	109	237	110	237	238	5	0,0
2	984	4.201	1.087	4.543	5.714	20	0,23
3	842	96.790	5.623	102.257	385.639	157	2,51
4	272	133.279	1.023	158.710	1.007.355	256	6,39
5	256	131.333	273	133.282	1.007.367	256	6,39

Table 3.3 – Mesures sans l'associativité de +

i	$\sharp\, eq_{th}$	$\sharp\, eq_{exec}$	$\sharp\, eq_{triv}$	$\sharp\, eq_M^+$	$\sharp\, eq_M^-$	$\sharp\, eq_N^+$	$\sharp\, eq_N^-$	δ_M^+	δ_M^-	δ_N^+	δ_N^-
1	140	140	12	96	3	125	0	2	1	3	0
2	462.020	5.378	1.333	1.504	240	3.805	43	2	129	3	423
3	320.011.850	1.349.453	990.374	27.122	9.150	349.929	4.329	2	439	3	2.774
4	199.408.295	4.405.368	3.791.217	13.168	7.762	606.389	5.441	2	156	3	12.551
5	1.136.712	20	4	0	4	12	4	0	8	1	947

Table 3.4 – Exécution des équations sans l'associativité de +

ensemble d'axiomes minimal pour le calcul booléen. Par exemple, cet ensemble reste complet si on lui retire l'équation qui exprime l'associativité de l'opérateur +. Nous donnons maintenant les mêmes mesures que précédemment concernant l'exécution de notre algorithme avec cet ensemble réduit d'équations. D'un côté, l'efficacité pourrait être améliorée puisqu'une équation est exécutée en moins, pour chaque valuation, mais, d'un autre côté, l'efficacité pourrait être diminuée parce que certaines fusions d'ensembles de structures ne seront plus réalisées ou seront réalisées plus tard, suite au retrait de l'axiome d'associativité de l'opérateur +. Les résultats expérimentaux sont donnés dans les tableaux 3.3 et 3.4. On remarque d'abord que le temps d'exécution est augmenté de 31 %. Le retrait de l'équation n'est donc pas bénéfique. Une comparaison des différents tableaux montre d'abord que trois fois plus de structures sont ajoutées à la collection, principalement lors de la quatrième itération. Moins d'équations sont exécutées mais plus d'équations font évoluer la collection de structures en lui ajoutant ou en lui supprimant des identifiants et des structures. On voit qu'une seule équation réduit le nombre de structures de 12.551, alors que la plus forte réduction est de 3.498 pour la première expérimentation. Cette réduction se produit aussi plus tôt. Globalement, le choix initial des équations permet une génération plus uniforme de la collection de structures, ce qui diminue le temps d'exécution et limite l'utilisation de la mémoire.

Nous analysons maintenant une autre variante dans laquelle le choix

i	M	N	M_{max}	N_{max}	N_{new}	$\sharp\, J$	t
0	5	5	5	5	5	0	0,00
1	91	192	92	192	193	5	0,00
2	720	3.235	801	3.472	4.245	17	0,16
3	735	39.956	3.573	41.015	89.812	75	1,44
4	261	129.520	1.956	130.910	366.905	246	5,00
5	256	131.333	331	132.484	375.074	256	5,42

Table 3.5 – Mesures pour le choix standard de l'identifiant à renommer

i	$\sharp\, eq_{th}$	$\sharp\, eq_{exec}$	$\sharp\, eq_{triv}$	$\sharp\, eq_M^+$	$\sharp\, eq_M^-$	$\sharp\, eq_N^+$	$\sharp\, eq_N^-$	δ_M^+	δ_M^-	δ_N^+	δ_N^-
1	175	110	9	80	2	99	0	2	1	3	0
2	402.136	4.543	1.505	1.099	205	2.833	35	2	86	3	269
3	188.181.149	230.438	153.430	10.229	2.955	74.053	755	1	2.374	2	8.404
4	199.930.500	7.399.145	7.128.280	7.605	7.622	263.243	4.736	1	1.183	2	15.689
5	1.469.250	960.285	952.147	51	142	7.996	136	1	4	2	300

Table 3.6 – Exécution des équations avec choix standard de l'identifiant à renommer

de l'identifiant à renommer, lorsqu'une opération *substitute* est exécutée, est effectué différemment. Supposons donc que l'algorithme applique une opération *substitute* à deux identifiants k et l. Le choix effectué dans notre algorithme sous étude consiste à conserver l'identifiant le plus ancien (selon la date de mise en service) et à renommer le plus récent. Ce choix assure qu'aucune valuation potentiellement nécessaire n'est écartée lors de la résolution des équations. Autrement dit, il assure que la i-ème itération de l'algorithme calcule effectivement une représentation de la congruence $\sim_i$ (voir le théorème 25) [4]. Une autre méthode de choix de l'identifiant à renommer, appelée ici "choix standard", consiste à conserver celui qui figure dans le plus de structures et à renommer l'autre. C'est la méthode que nous avons utilisée pour la résolution d'équations entre termes clos. Nous analysons les performances de l'algorithme avec cette méthode de choix et la présentation formelle de départ (incluant l'associativité de $+$). Les résultats sont présentés dans les tables 3.5 et 3.6. Nous voyons que les performances sont un peu moins bonnes : le temps d'exécution est augmenté de 11 % mais l'utilisation de la mémoire en termes de nombre de structures est très semblable. L'examen de la colonne $\sharp\, J$ suggère que beaucoup de valuations nécessaires au calcul complet de

4. Ceci suppose que nous générons effectivement toutes ces valuations, mais nous n'en générons en fait qu'une partie (voir plus bas).

$\sim_i$ ne sont pas utilisées : elles sont en fait "reportées" à l'itération suivante. (On voit aussi, à la ligne $i = 1$, de la table 3.6, que seulement 110 couples équation-valuation sont exécutés au lieu de 175, à la table 3.2.) Il s'en suit que les classes d'équivalences de la congruence finale sont fusionnées plus tard et que le travail à faire pour réaliser ces fusions est plus important. Par exemple, pour un certain couple équation-valuation, le nombre d'identifiants fusionnés s'élève à 15.689 au lieu de 3.498 au maximum, avec la méthode utilisée de préférence. Et, à nouveau, cette exécution de l'opération *normalize* se produit plus tard.

$x\,\overline{x} = 0$	$x\,(y+z) = x\,y + x\,z$
$x + \overline{x} = 1$	$x + y\,z = (x+y)\,(x+z)$
$x\,1 = x$	$(x+y)\,z = x\,z + y\,z$
$1\,x = x$	$x\,y + z = (x+z)\,(y+z)$
$x + 0 = x$	$(x\,y)\,z = z\,(x\,y)$
$0 + x = x$	$(x+y)+z = z+(x+y)$

Figure 3.2 – Axiomes de Vidiani pour le calcul booléen

Nous donnons enfin des résultats relatifs à une dernière expérience. Nous appliquons l'algorithme à l'ensemble d'équations proposé à la figure 3.2. Cet ensemble d'équations, proposé par Vidiani [Vid91], constitue un ensemble d'équations minimal pour le calcul booléen. (Il faut en exclure les deux équations écrites en traits grisés.) Si on applique notre algorithme à cet ensemble minimal, le programme se termine en erreur par manque de mémoire. (Toutefois, il se termine correctement, après $2,21$ secondes, si on limite à deux le nombre de constantes dans C.) Par contre, l'algorithme se termine si on ajoute aux axiomes de Vidiani les deux équations en traits grisés. Les résultats correspondants sont donnés dans les tables 3.7 et 3.8.

On constate que les résultats sont nettement moins bons qu'avec l'ensemble d'équations de la figure 3.1 : les axiomes choisis créent des structures représentant beaucoup de grands termes, qui se simplifient plus difficilement. Les deux équations que nous avons rajoutées servent à compenser une incomplétude potentielle de notre algorithme. En effet, nous avons choisi de n'engendrer qu'une partie des valuations pour limiter le nombre d'équations "trivialement vérifiées". Pour chaque identifiant i à considérer lors d'une itération, nous considérons seulement les groupes de valuations de la forme $\{x_1 \mapsto i,\ x_2 \mapsto i_2,\ \ldots,\ x_n \mapsto i_n\}$ où les identifiants sont tels que $i_2 \leq i_3 \leq \ldots \leq i_n \leq i$. Le nombre de ces valuations

i	M	N	M_{max}	N_{max}	N_{new}	$\sharp\, J$	t
0	5	5	5	5	5	0	0,00
1	250	490	251	490	491	5	0,01
2	2.524	9.070	3.890	13.566	21.231	24	0,64
3	1.986	111.386	6.615	122.346	373.351	160	3,67
4	352	131.525	6.710	153.546	3.007.471	256	20,91
5	256	131.333	7.923	155.488	5.202.143	256	30,83

Table 3.7 – Mesures : axiomes de Vidiani

i	$\sharp\, eq_{th}$	$\sharp\, eq_{exec}$	$\sharp\, eq_{triv}$	$\sharp\, eq_M^+$	$\sharp\, eq_M^-$	$\sharp\, eq_N^+$	$\sharp\, eq_N^-$	δ_M^+	δ_M^-	δ_N^+	δ_N^-
1	240	240	0	201	0	240	0	3	0	4	0
2	15.814.260	23.963	9.885	7.122	810	13.268	239	3	1.018	4	3.274
3	16.098.450.000	4.282.674	4.007.783	90.357	14.672	260.219	4.384	3	2.697	4	7.929
4	7.840.847.652	14.780.746	13.044.449	925.936	34.118	1.702.179	11.224	3	6.059	4	21.701
5	27.012.864	1.664.752	216.334	771.195	28.926	1.419.492	5.394	3	7.232	4	22.861

Table 3.8 – Exécution des équations pour les axiomes de Vidiani

est égal à C_{m+n-1}^n où m est le nombre d'identifiants i' utilisés par la collection tels que $i' \leq i$. Pour être certain que l'algorithme est complet, il faudrait considérer toutes les séquences telles que $i_2, i_3, \ldots i_n \leq i$, sans restrictions. Leur nombre est m^n qui est à peu près $n!$ fois plus grand que C_{m+n-1}^n. Dans le cas présent, n vaut au maximum 3, donc il y aurait à peu près 6 fois plus de valuations à considérer. Nous avons constaté expérimentalement que cette limitation du nombre de valuations ne détruit pas la complétude pour l'ensemble d'équations initialement choisi. Pour les axiomes de Vidiani, sans être sûr que la complétude est perdue d'un point de vue théorique, l'algorithme ne peut s'exécuter complètement. L'ajout des deux équations en grisé a pour but de "simuler" les valuations qui manquent en permutant les identifiants dans les structures. Ce n'était pas nécessaire pour l'ensemble initial d'équations qui contient des équations exprimant l'associativité et la commutativité des opérateurs . et +. D'ailleurs, on peut remplacer les deux équations en grisé par deux équations exprimant seulement la commutativité de . et +. Mais alors, l'exécution prend 133 secondes au lieu de 30,83. Ces axiomes sont donc moins efficaces. Signalons finalement que l'algorithme sort aussi de la mémoire si on l'exécute avec la version des axiomes de Vidiani présentée à la figure 3.2 (avec les équations grisées) mais en utilisant la version standard du choix de l'identifiant à remplacer.

3.2.5 Conclusion

Nous avons proposé un algorithme pour calculer, sous forme d'une collection de structures, la congruence définie par un ensemble fini d'équations non nécessairement closes Eq et un ensemble fini de constantes C. Cet algorithme se termine, du moins en théorie, si le nombre de classes d'équivalence de la congruence est fini. Une fois la collection de structures calculée, on peut l'utiliser pour minimiser, en temps linéaire, toute expression représentée dans la collection de structures. Il suffit de calculer l'identifiant de l'ensemble de structures correspondant à la classe d'équivalence de l'expression, puis d'y choisir une expression minimale comme expliqué à la section 3.1.2.

Nous avons appliqué avec succès cet algorithme pour calculer une collection de structures représentant toutes les formules booléennes (utilisant les trois opérateurs classiques de négation, conjonction et disjonction, ainsi que les constantes 0 et 1) construites à partir de trois lettres a, b et c. Nous avons aussi utilisé cet exemple pour discuter de l'applicabilité de l'algorithme et de certains choix cruciaux pour cette applicabilité tels que le choix d'équations "efficaces", la génération d'un ensemble "suffisant" de valuations et la bonne méthode de choisir l'identifiant à renommer lors de l'exécution d'une opération *substitute*. Dans le cas des formules booléennes, la plus grande limitation de notre méthode est la taille de la collection de structures à construire. En général, pour m lettres, elle comporte 2^{2^m} identifiants et donc $2^{2^{2m}+1} + 2^{2^m} + m + 2$ structures. Par exemple, pour $m = 4$, on arrive déjà à 65.536 identifiants et 8.590.000.134 structures, ce qui ne tient pas en mémoire, du moins avec notre implémentation utilisant des tableaux Java. Certains principes exposés dans cette section restent néanmoins applicables à la simplification des expressions si on évite de calculer une représentation complète de toutes les expressions possibles et qu'on cherche à réaliser un algorithme "guidé par la forme de l'expression à simplifier". Nous proposons de tels algorithmes dans la section suivante.

3.3 Simplification des Expressions

Nous abordons maintenant le problème qui a motivé et a été à l'origine de tout notre travail : la simplification des expressions dans le sens général du terme, indépendamment d'une forme particulière d'expressions. Nous allons donc proposer une méthode générale de simplification d'expressions basée sur l'utilisation des collections de structures. Toutefois, nous utilisons une fois de plus le cadre des formules booléennes pour illustrer son applicabilité pratique. Nous terminons cette section en suggérant des pistes d'amélioration de la méthode proposée.

Avant de développer ces points, il est utile de remarquer que le problème traité dans cette section est davantage un problème "pratique" que les problèmes abordés précédemment. La simplification des expressions en général est un problème qui échappe à toute définition complètement satisfaisante. Nous suivons donc une démarche plus pragmatique que dans les sections précédentes sans rechercher à établir des résultats "formels" mais en montrant expérimentalement l'intérêt de la méthode que nous proposons.

3.3.1 Intérêt des collections de structures pour la simplification

Le problème général de la simplification des expressions (voir, par exemple, [Knu97], pages 339 et 346) se traite généralement en simplifiant d'abord les sous-expressions de l'expression à simplifier puis en énumérant diverses transformations de l'expression obtenue et en en choisissant une "plus simple" parmi celles-ci. C'est une méthode récursive basée sur des choix locaux qui ne sont plus "remis en question" par la suite. Le caractère "compositionnel" de cette approche limite sa puissance théorique. Une approche plus ambitieuse consiste à rechercher, parmi toutes les expressions formellement égales à l'expression à simplifier, celle qui est la plus simple, selon le critère de simplicité choisi. Et il se fait qu'il est souvent impossible de déterminer une telle expression sans en construire plusieurs autres qui sont plus compliquées, voire beaucoup plus compliquées, que l'expression de départ. Un exemple très simple est le suivant. Supposons qu'on veuille simplifier la formule booléenne $a + ab$, en utilisant les règles de base du calcul booléen exprimées par les équations de la figure 3.1. On écrira, par exemple :

$$
\begin{array}{rcll}
a + a\,b & = & \overline{\overline{a}} + \overline{\overline{a\,b}} & (\overline{\overline{x}} = x\ (2\times)) \\
& = & \overline{\overline{a}\,\overline{(ab)}} & (\overline{x\,y} = \overline{x} + \overline{y}) \\
& = & \overline{\overline{a}\,(\overline{a} + \overline{b})} & (\overline{x\,y} = \overline{x} + \overline{y}) \\
& = & \overline{(\overline{a} + 0)(\overline{a} + \overline{b})} & (x + 0 = x) \\
& = & \overline{\overline{a} + 0\ \overline{b}} & (x + y\,z = (x + y)\,(x + z)) \\
& = & \overline{\overline{a} + \overline{b}\ 0} & (x\,y = y\,x) \\
& = & \overline{\overline{a} + 0} & (x\,0 = 0) \\
& = & \overline{\overline{a}} & (x + 0 = x) \\
& = & a & (\overline{\overline{x}} = x\ )
\end{array}
$$

On remarque bien que presque toutes les formules intermédiaires sont passablement plus compliquées que la formule à simplifier, de départ. On remarque aussi que trois des équations doivent être appliqués “dans les deux sens” et, deux fois, d'abord, pour obtenir une expression plus compliquée et, ensuite, pour en obtenir une plus simple [5]. On remarque enfin que les équations sont généralement instanciées “à l'intérieur” d'expressions plus compliquées, ce qui signifie qu'on applique presque toujours, implicitement, une règle d'inférence qui exprime que l'égalité entre les expressions est une congruence.

L'utilisation des collections de structures permet de réaliser automatiquement toutes les inférences de l'exemple ci-dessus, de manière beaucoup plus simple et efficace qu'en écrivant un algorithme qui énumère explicitement tous les termes possibles et leur applique les équations. En effet, sur cet exemple, il suffit d'abord de créer une collection de structures représentant l'expression $a + ab$, puis de résoudre toutes les équations de la figure 3.1 par rapport à toutes les valuations qu'on peut construire à partir des identifiants de cette collection. Il est toutefois nécessaire de refaire le processus d'application des équations une seconde fois car la première application ne construit pas suffisamment d'expressions à partir de l'expression de départ et des axiomes. (On peut cependant enrichir l'ensemble des équations pour obtenir le résultat en une seule étape. Nous utilisons un tel ensemble “enrichi” à la section 3.3.3.) On obtient alors la simplification de $a + ab$ en a, en déterminant une expression de taille minimale, équivalente à $a + ab$, dans la collection de structures.

5. Ce qu'on ne peut pas faire, par exemple, avec des règles de réécriture.

Nous allons maintenant expliquer comment nous mettons en œuvre pratiquement la méthode suggérée ci-dessus.

3.3.2 Algorithmes de simplification

Le principe de notre méthode de simplification peut s'exprimer dans les quelques points suivants.

1. Nous exprimons la connaissance que nous avons du domaine "logico-mathématique" par rapport auquel nous voulons simplifier des expressions, sous forme d'un ensemble d'équations non (nécessairement) closes Eq [6].
2. Pour chaque expression que nous désirons simplifier, nous créons d'abord une collection de structures représentant cette expression (et donc aussi toutes ses sous-expressions).
3. Nous résolvons les équations Eq par rapport aux valuations pouvant être construites à partir des identifiants de la collection de structures, comme expliqué à la section 3.2.2.
4. Nous déterminons la taille d'une expression minimale de la collection de structures équivalente à l'expression de départ.
5. Si la taille obtenue est satisfaisante ou si la collection de structures n'a pas été modifiée par la dernière exécution du point 3, l'algorithme se termine. Sinon, il reprend au point 3, avec les identifiants de la collection de structures modifiée (augmentée).

La mise en œuvre des principes ci-dessus se heurte à de nombreuses difficultés pratiques que nous passons en revue maintenant et pour lesquelles nous proposons chaque fois des solutions "empiriques".

1. Si, à chaque étape de l'algorithme suggéré ci-dessus, nous résolvons les équations avec toutes les valuations possibles, la taille de la collection de structures va croître très rapidement, augmentant énormément le nombre de valuations à l'étape suivante, et ainsi de suite. L'algorithme ne peut donc se terminer, par manque de mémoire, dans la plupart des cas. Nous proposons de limiter le nombre de valuations en engendrant seulement celles d'entre elles qui sont "en relation" avec l'expression à simplifier. Une telle valuation appliquée à une équation doit idéalement donner lieu à des structures correspondant à des sous-termes de l'expression courante à

6. Notre approche peut aussi s'appliquer à des règles de simplification plus élaborées, intégrant, par exemples, des préconditions d'applicabilité. Mais nous n'avons pas, à ce jour, réalisé d'algorithmes intégrant ces possibilités plus générales.

simplifier, pour créer de nouvelles sous-expressions équivalentes et, finalement, en produire de plus simples.

Pour nous limiter à de telles valuations, nous pouvons envisager de procéder par "pattern matching" des parties gauches et droites des équations avec les structures de la collection et les structures accessibles à partir de celles-ci. C'est une approche descendante et compliquée à mettre en œuvre, notamment parce que la collection de structures se modifie en cours de route[7]. Nous proposons une méthode ascendante, beaucoup plus simple, et qui donne d'assez bons résultats pour les exemples que nous avons traités (voir section 3.3.3). Cette méthode ne permet toutefois pas de traiter des équations comportant plus de trois variables. D'autres "tactiques" devront donc être imaginées pour étendre notre méthode à plus de trois variables. Nous procédons comme suit, à chaque itération.

(a) Nous résolvons d'abord toutes les équations n'utilisant qu'une seule variable x_1, par rapport à toutes les valuations $\{x_1 \mapsto i\}$ telles que i est un identifiant de la collection, au début de l'itération courante.

(b) Nous résolvons ensuite toutes les équations utilisant deux variables x_1, x_2 par rapport à toutes les valuations $\{x_1 \mapsto i_1, x_2 \mapsto i_2\}$ telles que $f(i_1, i_2) : i$ est une structure de la collection, au début de l'itération courante.

(c) Enfin, et c'est le cas le plus compliqué et le plus fréquent, nous résolvons toutes les équations utilisant trois variables x_1, x_2, x_3 par rapport à toutes les valuations $\{x_1 \mapsto i_1, x_2 \mapsto i_2, x_3 \mapsto i_3\}$ telles qu'il existe deux structures de la forme $f(i_1, i_2) : i$ et $f(i, i_3) : i'$ ou deux structures de la forme $f(i_1, i) : i'$ et $f(i_2, i_3) : i$, au début de l'itération courante.

Ces façons de choisir les valuations n'assurent pas que les résolutions d'équations vont toutes produire des structures "utiles" pour la simplification mais elles assurent, en tous cas, que les identifiants placés dans les valuations sont "proches les uns des autres" dans la collection de structures et donc que la résolution des équations avec ces valuations va modifier la collection de structures "à cet endroit de la collection".

Cette méthode de choix est aussi efficace à mettre en œuvre avec les structures de données de bas niveau utilisées pour réaliser les collections de structures. Par exemple, pour déterminer toutes les

7. Nous avons réalisé un algorithme basé sur cette idée mais il n'a pas donné de très bons résultats.

valuations $\{x_1 \mapsto i_1, x_2 \mapsto i_2, x_3 \mapsto i_3\}$ telles qu'il existe deux structures de la forme $f(i_1, i_2) : i$ et $f(i, i_3) : i'$, il suffit, pour chaque identifiant i, de parcourir la liste $sameId(i)$ (voir section 2.4.2) pour obtenir toutes les structures $f(i_1, i_2) : i$, puis la liste $sameId_1(i)$ pour obtenir les structures $f(i, i_3) : i'$. Le nombre total de structures à retrouver est de l'ordre de $\frac{N^2}{M}$ où M est le nombre d'identifiants et N le nombre de structures, au début de l'itération courante. Si la collection est creuse, ce nombre est de l'ordre de M, ce qui est nettement inférieur au nombre M^3 de valuations possibles.

2. Lorsqu'une valuation est choisie, nous avons encore plusieurs choix possibles pour la résolution des équations par rapport à cette valuation. Une première solution, qui semble raisonnable, est de vérifier qu'un, au moins, des deux identifiants obtenus en appliquant l'opération *toSet* aux parties gauche et droite de l'équation appartenait déjà à la collection au début de l'itération courante et de ne pas leur appliquer l'opération *unify*, sinon. Ce choix que nous appelons *circonspect*, assure que toute expression, représentée dans la collection, est sous-expression d'au moins une expression équivalente à l'expression à simplifier. On n'engendre pas d'expression n'ayant "rien à voir" avec l'expression à simplifier. Un autre choix, que nous appelons *offensif*, consiste à résoudre l'équation de toute façon, en pariant que les termes rajoutés à la collection seront mis en lien tôt ou tard avec des sous-termes de l'expression à simplifier. Cette solution utilise plus de mémoire mais peut éventuellement conduire à une meilleure simplification ou même à une simplification plus rapide.

3. Quelle que soit la méthode utilisée pour choisir les valuations et les équations à résoudre effectivement, il arrive fréquemment que la mémoire soit complètement utilisée avant la fin d'une itération. Nous avons alors plusieurs options pour pouvoir continuer la simplification.

   La première chose à faire, de toute façon, est de "récupérer" de la mémoire. Il est aisé, avec les structures de bas niveau utilisées pour réaliser les collections de structures, de parcourir les ensembles de structures et d'en retirer celles qui semblent les moins utiles. Nous avons réalisé quatre algorithmes de récupération de mémoire. Le plus simple, appelons-le gc_0, vide complètement la collection de structures. Il faut donc sauver une copie d'une expression minimale courante, avant de l'utiliser. Un autre, appelons-le gc_1, garde toutes les structures des ensembles de structures représentant des

sous-expressions d'au moins une expression équivalente à une expression minimale courante, et supprime les autres. Cet algorithme enlève aussi de la collection toutes les structures contenant un identifiant i tel que tous les termes représentés par cet identifiant ont une taille supérieure à celle de l'expression minimale courante. Un troisième algorithme, appelons-le gc_2, ne conserve que les structures représentant des sous-expressions d'une expression minimale. Finalement, le quatrième algorithme, gc_3, fait comme gc_2, sauf qu'il ne conserve qu'une seule structure, pour chaque ensemble de structures E_i. Il ne maintient donc, dans la collection de structures, qu'une seule expression minimale. Munis de ces différents algorithmes, nous pouvons faire différents choix si la mémoire vient à manquer au cours d'une itération.

(a) Nous pouvons appliquer l'algorithme gc_1 chaque fois que la mémoire est pleine et continuer l'itération courante jusqu'à ce qu'aucune récupération ne soit possible.

(b) Une autre façon est d'appliquer gc_1 chaque fois que la taille de l'expression minimale diminue, puis de continuer l'itération courante.

(c) Une troisième façon consiste à abandonner l'itération courante.

À la fin de chaque itération, nous devons choisir entre deux possibilités.

(a) La première possibilité consiste à itérer sur base de la collection de structures courante. Nous le faisons toujours si la taille des expressions minimales courantes est strictement inférieure à leur taille avant l'itération. Car cette diminution suggère qu'il y a encore "du potentiel" dans la collection de structures courante. Rappelons que la nouvelle itération va utiliser tous les nouveaux identifiants et les structures qui ont été ajoutées à la collection durant la précédente itération. Si la stratégie *circonspecte* est utilisée pour la résolution des équations, nous reprenons avec la collection de structures telle qu'elle. Si c'est la stratégie *offensive* qui est utilisée, nous exécutons d'abord l'algorithme gc_1. Donc, quelle que soit la stratégie, on se retrouve au début de la nouvelle itération avec uniquement des structures représentant des sous-expressions d'expressions équivalentes aux expressions minimales courantes.

(b) La seconde possibilité consiste à repartir sur de nouvelles base en nettoyant drastiquement la collection de structures : soit en

appliquant un des deux algorithmes gc_2 ou gc_3 avant d'itérer à nouveau, soit en appliquant l'algorithme gc_0, après avoir pris une copie d'une expression minimale courante, puis en reprenant tout au départ avec cette nouvelle expression à minimiser.

4. Il est impossible de mettre au point une stratégie infaillible pour choisir entre les deux possibilités ci-dessus, dans le cas où la taille des expressions minimales n'a pas diminué lors de la dernière itération. Une bonne stratégie peut éventuellement être choisie en fonction d'un type de problème. Dans notre application à la simplification des formules booléennes, si la taille n'a pas diminué, nous itérons au maximum trois fois et seulement deux fois si les deux itérations n'ont pas donné lieu à un dépassement de mémoire. Cette méthode de choix est basée sur les expérimentations réalisées.
5. Il est tout aussi impossible de trouver un critère imparable pour arrêter définitivement l'algorithme : en général, décider si une expression est la plus simple possible, ou non, est un problème extrêmement difficile (voir [MS72]). Des solutions pragmatiques sont, par exemple, d'arrêter l'algorithme si la taille n'a pas diminué après un nombre fixé d'itérations ou si le temps d'exécution a dépassé une borne fixée *a priori*.
6. Pour augmenter la rapidité de l'algorithme, on peut aussi jouer sur la quantité de mémoire effectivement allouée à la collection de structures. C'est le cas lorsque les expressions à simplifier de départ comportent "beaucoup d'opportunités de simplification". En effet, on peut à tout moment choisir entre continuer avec la collection de structures courantes ou en extraire une expression minimale courante, vider la mémoire et reprendre avec la nouvelle expression. En démarrant l'algorithme avec relativement peu de mémoire, on sera conduit plus tôt à faire le second choix. On peut alors augmenter la quantité de mémoire allouée avant de reprendre la simplification. De la sorte, on avancera plus vite dans le processus de simplification mais on manquera peut-être certains choix plus pertinents car en démarrant avec plus de mémoire on aurait trouvé une meilleure expression minimale intermédiaire (au prix de plus de temps de calcul).

3.3.3 Simplification des formules booléennes

Nous donnons maintenant des résultats expérimentaux obtenus, pour la simplification des formules booléennes, avec un algorithme basé sur les principes exposés à la section précédente. Nous utilisons les équations présentées à la figure 3.3. Cet ensemble d'équations est évidemment redondant mais il a été choisi ainsi pour diminuer le nombre d'itérations à effectuer par l'algorithme.

Il n'est toutefois pas certain qu'il n'existe pas d'ensemble nettement plus simple qui lui soit préférable.

$\overline{0}$	$=$	1	$x\,y$	$=$	$y\,x$
$\overline{1}$	$=$	0	$x+y$	$=$	$y+x$
$x\,0$	$=$	0	$(x+y)+z$	$=$	$x+(y+z)$
$0\,x$	$=$	0	$(x\,y)\,z$	$=$	$x\,(y\,z)$
$x+0$	$=$	x	$\overline{x\,y}$	$=$	$\overline{x}+\overline{y}$
$0+x$	$=$	x	$\overline{x+y}$	$=$	$\overline{y}\,\overline{x}$
$x+1$	$=$	1	$x+y\,z$	$=$	$(x+y)\,(x+z)$
$1+x$	$=$	1	$x\,y+z$	$=$	$(x+z)\,(y+z)$
$\overline{x}\,x$	$=$	0	$(x+y)\,z$	$=$	$x\,z+y\,z$
$x+\overline{x}$	$=$	1	$x\,(y+z)$	$=$	$x\,y+x\,z$
$\overline{x}+x$	$=$	1	$(x\,y)\,z$	$=$	$z\,(x\,y)$
$x\,\overline{x}$	$=$	0	$(x+y)+z$	$=$	$z+(x+y)$
$\overline{\overline{x}}$	$=$	x			

Figure 3.3 – Équations pour la simplification des formules booléennes

Nous avons appliqué l'algorithme à différentes formules booléennes, écrites sous forme quelconque, générale, non mises sous forme disjonctive. Pour chaque formule, nous exécutons l'algorithme avec chacune des stratégies *offensive* et *circonspecte* et nous mesurons la taille des formules obtenues à certaines étapes de l'exécution. Nous calculons aussi, avec un des algorithmes présentés au chapitre 1, la liste des impliquants premiers de ces formules et nous leur appliquons les mêmes algorithmes et fournissons les mêmes mesures. Nous commençons par la formule *F16*, présentée à la figure 3.4. Cette formule utilise 16 lettres différentes et sa

taille est égale à 800 (parenthèses non comprises). Elle constitue donc un problème assez difficile *a priori*. La figure présente aussi la liste des impliquants premiers de cette formule (*IP16*) et la formule équivalente la plus courte que notre algorithme ait pu calculer (*FS16*). Nous notons la négation avec le caractère `!`, pour la facilité.

F16	`(!(!m+!a)+(i!kde(e+!a)+(j+!b)!d)!n(!!oo+n(j+!!(l+c))(!!c+!kn)+!!dk(a+e)+!n)+g)(!!((!k!o+!!m+p(i+m)!i)(!l+!b)p(k+!h)k!((p+h)!k))!(d!!!((!((g+o)e)+!(j!! !o)p+!j+g+me)(g+lp)))+(!p+l+i)!!(b(!c+j+!d+k)(ab+! (!(f+!b)+f)))(k+!((ceji+!c+i+g+bm)!!kdn+!((m+c+!d+ !jm)(!p+!p)))+!!e(l!i+n)!m!(!k+n))(!((a+j)!n!k+!a) mg!g+d+!e+ep)(p+f)!cdo!((!lg+m+p+!d)djcim(!e+i)m+k !(fh)+!i+hf(!a+p)+!(fbh)!(!!!(n!lp)+b))(b+g))((!!((k(!h+ne)+!!a+f+h)(hd+!h+(!c!m+!a)(f+b!b))(b(j+ni) +a(f+n)+!(a(f+!o))(f+i)!((!n+o)e)!(j+ol+((o+d+g+i) !!b!o+!j)!p))!((!h!n+!!(bp)(e+!g(fka+k)+b))(!(!gb+ bm)+a+!(!ei)+f+m!j+!((l+b)m+f)(b+d)+!(b+g+f+e)+p+! !c+!o)))+o+g+!g+g!h+(h+dd)((!e!g+l+g)(o+l+!m)+(!jm f+e+n)(m+c)+a)+(!d+!(n+!g)(m+b)(b+!(c+!b)(h+!j+d)))!d!l(i(m+!a+d)+h)+!(!p(!i+!h))+!o+h+i+f!n!c(b+d+o +pc+k)+ba+!g+!p)!d!h(!f+k+o)(d+f+!k)!!(b+!e(!a+g+f)kp+!k+!((c+j)!k(d+!c)))(p!mg+!(!j+!(fd+!h)))+!h+! d+(!a+n+!b+!b+m)!(k+!h))`
IP16	`ab!cdfi!kmo!p + !b!dkm!np + a!b!dkmp + a!dk!lmp + !b!dgkmp + !bg!hkmp + !dgk!lmp + g!hk!lmp + !djk!lm!np + a!b!hklmp`
FS16	`(abdifo!(p + c + k) + kp(al + (a + (!b + j)!n)!d + g)!(hd + bl))m`

Figure 3.4 – Une formule de taille 800, utilisant 16 lettres différentes

Nous donnons, à la table 3.9, une comparaison des tailles des formules minimales courantes, en fonction du temps d'exécution dépensé par l'algorithme de minimisation. La colonne t donne les temps d'exécution, en secondes, les colonnes $F16_o$, $F16_c$, $IP16_o$ et $IP16_c$ fournissent la taille obtenue pour la formule réduite après cette durée d'exécution, en appliquant l'algorithme à la formule *F16* ou à la formule *IP16* avec soit la stratégie offensive ($_o$), soit la stratégie circonspecte ($_c$). Les valeurs des tailles sont omises si le fait d'allouer plus de temps d'exécution ne

permet pas de diminuer la taille de l'expression réduite. Pour la formule originale *F16*, la stratégie circonspecte permet une réduction plus rapide au début, sans doute parce qu'elle nécessite moins de mémoire pour être effective. Nous allouons en effet la mémoire progressivement, en l'augmentant lorsqu'une ou plusieurs itérations ne produisent plus de réduction de taille. Lorsque la taille est réduite à 100 symboles, la stratégie offensive permet une meilleure réduction : la totalité de la mémoire est alors allouée. On constate aussi que, comme prévu, la réduction n'est (relativement) rapide que durant les premières secondes d'exécution. Quand les formules minimales courantes sont relativement réduites, la stratégie de recentrer la recherche sur les structures représentant uniquement de telles formules ne produit plus d'effet et le problème devient purement "combinatoire". Nous observons également que le fait de partir de la liste des impliquants premiers ne fournit pas de meilleurs résultats bien qu'il permette une réduction plus rapide. C'est relativement logique puisque la formule *IP16* ne contient que 154 symboles. On voit toutefois que, cette formule étant déjà relativement réduite, l'algorithme prend du temps avant d'obtenir une bonne réduction.

t	$F16_o$	$F16_c$	$IP16_o$	$IP16_c$
0	800	800	154	154
0,5	758	758	154	154
1	758	758	154	154
2,5	688	275	151	154
5	595	263	132	141
10	237	193	121	129
25	151	123	86	108
50	108	107	81	100
100	101	102	78	94
250	87	98	64	90
500	85	95	60	88
1.000	69	91	54	86
2.000	53	72		
3.000	52	70		

Table 3.9 – Taille des expressions réduites après t secondes (formule *F16*)

Nous donnons maintenant un autre exemple de formule de taille 800, utilisant 16 lettres. Les résultats expérimentaux sont montrés à la figure 3.10. Une grande différence entre cet exemple et le précédent est que la

t	$F16'_o$	$F16'_c$	$IP16'_o$	$IP16'_c$
0	800	800	928	928
0,5	772	772	928	928
1	772	772	928	928
2,5	730	522	883	928
5	636	492	863	914
10	530	461	836	878
25	416	363	771	860
50	414	343	764	846
100	335	340	726	791
250	157	336	702	739
500	142	331	643	656
1.000	139	320	591	609
2.000	132	318	400	533
3.000		317	380	513

Table 3.10 – Taille des expressions réduites après t secondes (formule $F16'$)

liste des impliquants premiers $IP16'$ est beaucoup plus grande et utilise 928 symboles, au lieu de 154, dans l'exemple précédent. Conséquemment, la réduction de la liste des impliquants premiers fournit de beaucoup moins bons résultats que celle de la formule originale.

Nous donnons maintenant d'autres résultats pour des formules choisies "aléatoirement", de taille 800 mais utilisant entre 3 et 15 lettres distinctes. Ces résultats sont donnés dans les tables 3.11, 3.12 et 3.13. Le nombre de lettres utilisées se déduit du nom de la formule. Par exemple, la formule *F5* utilise 5 lettres distinctes. On remarque que les temps d'exécution nécessaires pour obtenir les formules de plus petite taille diminuent rapidement lorsque le nombre de lettres diminue. On constate également que les temps sont meilleurs lorsqu'on fait la simplification en partant de la liste des impliquants premiers, du moins lorsque la taille de cette liste est significativement plus petite que celle de la formule de départ. Ce n'est pas forcément le cas : la liste des impliquants premiers peut être beaucoup plus grande que celle de la formule de départ. Pour la formule *F15*, la liste des impliquants premiers est 1,5 fois plus longue que la formule elle-même et aucune des deux stratégies ne conduit à une réduction satisfaisante dans le temps limite de 3.000 secondes fixé pour l'exécution des algorithmes.

t	$F15_o$	$F15_c$	$IP15_o$	$IP15_c$
0	800	800	1.231	1.231
0,5	748	748	1.231	1.231
2,5	676	577	1.214	1.231
5	357	568	1.204	1.217
7,5	286	568	1.194	1.207
10	282	495	1.173	1.199
25	186	234	1.140	1.178
50	140	177	1.083	1.158
75	138	170	1.078	1.136
100	138	170	1.037	1.065
250	137	152	989	997
500	135	142	954	939
750	133	140	923	865
1.000	133	140	917	853
2.000	130	138	850	772
3.000	128	138	731	723

t	$F9_o$	$F9_c$	$IP9_o$	$IP9_c$
0	800	800	55	55
0,5	672	672	55	55
2,5	670	648	29	54
4	582	648	28	48
5	569	639	28	46
7,5	524	631	28	41
10	494	550	28	38
12,5	482	529	27	38
15	450	506	27	34
16	450	494	27	29
20	369	465	27	28
50	176	133	25	26
75	85	131		
110	55	125		
345	25	67		
800		26		

Table 3.11 – Taille des expressions réduites après t secondes (formules *F15* et *F9*)

t	$F7_o$	$F7_c$	$IP7_o$	$IP7_c$
0	800	800	73	73
0,5	722	722	73	73
2,5	720	376	51	70
4	217	373	36	58
5	166	373	29	46
6	140	199	29	37
7	63	68	27	31
8	36	65	27	31
9	36	60	24	27
10	35	47	24	27
12,5	28	45	24	26
15	24	37	22	
17,5	24	31		
20	24	28		
22,5	24	26		
30	22			

t	$F6_o$	$F6_c$	$IP6_o$	$IP6_c$
0	800	800	28	28
0,5	726	726	28	28
1	726	726	22	25
2,5	615	357	18	25
4	591	191	13	19
5	341	161		13
7,5	208	93		
10	72	59		
12,5	40	42		
15	33	38		
16	32	38		
20	29	35		
50	13	20		
75		18		
80		13		

Table 3.12 – Taille des expressions réduites après t secondes (formules *F7* et *F6*)

Le cas de la formule *F3* nécessite quelques explications supplémentaires. Nous n'avons pas utilisé une formule choisie "au hasard" car la plupart

t	$F5_o$	$F5_c$	$IP5_o$	$IP5_c$
0	800	800	25	25
0,5	669	669	25	25
1	644	644	11	22
2,5	599	582		22
3	438	582		22
4	392	403		11
5	12	378		
7,5	11	355		
10		328		
12,5		35		
15		12		
16		11		

t	$F3_o$	$F3_c$	$IP3_o$	$IP3_c$
0	34	34	29	29
0,4	28	34	29	29
0,5	24	28	29	29
0,6	15	15	20	26
0,7	15	15	16	26
0,8	15	15	12	26
1,1	12	15		26
3		12		26
3,2				24
3,3				22
3,4				16
3,5				12

Table 3.13 – Taille des expressions réduites après t secondes (formules *F5* et *F3*)

de ces formules sont trop faciles à simplifier. Nous avons choisi la formule

```
(!(b+c)+(a+b)c+(a+b))(!(b+c)+(a+b)c+(!(a+c)+ac))
```

équivalente à la formule minimale :

```
(b + a)c +!(c + ab)
```

Nous avons vérifié, en utilisant l'algorithme de minimisation stricte présenté à la section 3.2, que cette formule est bien minimale. On constate que le temps nécessaire à la simplification, par les deux versions de l'algorithme de cette section, est significativement inférieur au temps nécessaire pour construire la collection de structures qui représente la totalité des formules booléennes utilisant au plus trois lettres. Toutefois, l'ordre de grandeur est à peu près le même. En réalité, si on n'y prend pas garde, l'algorithme de cette section-ci qui utilise la stratégie *offensive*, peut calculer l'entièreté de la collection de structures représentant toutes les formules. Pour éviter ce problème nous limitons à deux le nombre d'itérations de l'algorithme lorsque la taille de la formule ne diminue pas et que l'itération ne se termine pas à cause d'un manque de mémoire. Nous limitons aussi l'application des équations aux valuations dont la somme des tailles des formules minimales représentées par les identifiants de la valuation reste inférieure à $1,5$ fois la taille de la formule minimale courante. C'est une restriction arbitraire qui peut évidemment empêcher l'algorithme de trouver une bonne solution dans d'autres cas de figure.

Nous clôturons cette liste d'exemples par deux formules supplémentaires. La formule *FSp* est choisie spécialement pour sa liste d'impliquants premiers, qui est à peu près 23 fois plus grande que la formule elle-même.

Sans surprise, les algorithmes appliqués à la liste des impliquants premiers donnent de très mauvais résultats. Nous donnons aussi la formule *F20* qui utilise 20 lettres distinctes. Les résultats pour cette formule sont excellents. Nous ne donnons pas les résultats pour la liste des impliquants premiers car la représentation des formules utilisée par nos algorithmes de calcul des impliquants premiers est limitée à 16 lettres (voir le chapitre 1).

t	FSp_o	FSp_c	$IPSp_o$	$IPSp_c$
0	95	95	2.174	2.174
0,5	95	95	2.174	2.174
2,5	58	86	2.149	2.174
5	44	56	2.108	2.174
7,5	44	50	2.108	2.141
10	44	48	2.077	2.141
25	44	40	2.063	2.138
50	42	38	2.051	2.112
75	42		1.982	2.104
100	42		1.970	2.069
175	38		1.967	2.061
500			1.849	1.785
1.000			1.699	1.605
2.000			1.583	1.342
3.000			1.321	1.236

t	$F20_o$	$F20_c$
0	800	800
0,5	742	742
2,5	696	614
5	545	527
7,5	533	516
10	507	511
25	457	161
50	407	158
75	370	156
100	67	79
250	39	72
500	34	46
750	34	37
1.000	33	36
1.150		33

Table 3.14 – Taille des expressions réduites après t secondes (formule *FSp*)

3.3.4 Conclusion et pistes d'améliorations et de travaux futurs

Dans cette section, nous avons présenté une méthode générale pour simplifier les expressions, basée sur les collections de structures. La méthode proposée est moins systématique que les algorithmes proposés dans le reste de ce travail. Elle utilise différentes heuristiques à choisir et à adapter en fonction des exemples à traiter. Le problème général de la simplification étant complètement "combinatoire", le point clé d'un bon algorithme est de réduire "l'espace de recherche" en y maintenant des solutions acceptables, sinon optimales. Nous le faisons en maintenant la taille des expressions minimales courantes et, lorsque cette taille diminue, en supprimant les structures correspondant à des expressions non minimales. Cette méthode permet d'obtenir rapidement des réductions

pour les expressions qui sont fortement simplifiables mais montre ses limites lorsque la taille des expressions minimales courantes se rapproche de celle d'une expression minimale théorique.

La méthode que nous avons proposée est *a priori* indépendante du problème particulier de la simplification des expressions booléennes. Il est possible que nous puissions l'améliorer dans le futur, pour de telles expressions seulement, en imaginant un algorithme qui applique une stratégie du type "diviser pour régner", semblable à la méthode de calcul des impliquants premiers de Coudert (voir la section 1.3.4 et aussi l'annexe A). Nous avons aussi l'intention d'adapter notre méthode au problème de la simplification des expressions régulières (ou rationnelles) (voir [Sto15]). Pour ce problème, il est nécessaire d'ajouter des conditions d'applicabilité aux équations. Il faudra donc étendre notre cadre "théorique".

Conclusion

Dans ce document présentant les résultats de notre recherche, nous avons introduit une nouvelle structure de données conçue dans le but de simplifier efficacement des expressions. Nous avons décrit précisément sa sémantique et son implémentation, nous avons estimé sa complexité théorique et nous avons évalué expérimentalement son efficacité pratique. Nous l'avons utilisée pour résoudre divers problèmes tels que la résolution d'équations entre termes clos ou le calcul de la congruence définie par un ensemble d'équations et un ensemble de générateurs, et bien sûr aussi, finalement, pour la simplification des expressions. Nous avons appliqué les méthodes obtenues aux formules booléennes, auxquelles nous avons aussi consacré un chapitre indépendant, relatif au calcul de la liste des impliquants premiers. Dans ce chapitre, nous avons présenté diverses méthodes proposées dans la littérature, dans un cadre unifié clarificateur, nous avons donné des preuves directes de leur correction et nous avons comparé leur efficacité expérimentalement.

Nous pensons avoir montré que les résultats que nous avons obtenus sont significatifs et qu'ils indiquent que la notion de collection structures fournit une base solide pour de nouvelles recherches à réaliser. Dans le domaine, initialement choisi par nous, de la simplification proprement dite des expressions, de nombreuses expérimentations nouvelles nous viennent à l'esprit. Tout d'abord, nous allons prochainement appliquer notre méthode au problème de la simplification des expressions régulières (ou rationnelles). Il n'existe pas de système fini d'équations caractérisant la théorie des expressions régulières mais il ne sera pas difficile d'étendre notre méthode pour prendre en compte la règle d'inférence supplémentaire dont a besoin pour caractériser cette théorie. Les méthodes de simplification que nous avons proposées à la section 3.3 seront donc applicables avec peu de changement. Par ailleurs, plus généralement, nous pourrons étendre notre méthode pour qu'elle soit capable de traiter des implications logiques liant des conjonctions d'équations (ou clauses de Horn). Plus généralement encore, nous pourrons traiter des

règles d'inférence comportant des conditions d'applicabilité quelconques (portant sur les termes représentés dans la collection de structures). Moyennant ces extensions de notre méthode, nous envisageons d'explorer l'applicabilité de celle-ci à toutes autres sortes d'expressions, incluant les expressions arithmétiques et algébriques, trigonométriques, etc.

L'étude des travaux reliés aux résultats présentés dans cette thèse, nous suggère aussi de nouveaux domaines d'application de notre méthode dans le domaine de la démonstration automatique de théorèmes. Tout d'abord, nous pourrions explorer l'utilisation des collections de structures dans le domaine de l'unification rigide, évoqué seulement à la section 2.6.3.2. Ainsi, l'article [BFR17] propose un ensemble de règles d'inférence (un "calcul") pour résoudre le problème de l'unification rigide. La difficulté d'utiliser les collections de structures pour mettre en œuvre ce calcul vient de la nécessité de procéder par essais et erreurs (retours en arrière) lorsqu'une piste de solution mène à un échec. Il nous faudrait donc généraliser notre implémentation actuelle des collections de structures pour prévoir des "points de reprise" permettant de restaurer la collection dans un état précédent. Une telle généralisation est certainement possible mais la façon exacte et la plus efficace de le faire requiert une nouvelle recherche.

D'autres applications, évoquées ci-dessous, nous ont été suggérées récemment par Pascal Fontaine.

Notre algorithme de calcul de congruence permet de représenter aisément l'ensemble des formules booléennes comportant au plus trois lettres distinctes et ainsi de minimiser en temps linéaire toutes les formules de cette sorte. Représenter les formules avec quatre lettres au plus requiert de créer 8.590.000.134 structures, ce qui est sans doute possible moyennant certaines modifications de nos structures de bas niveau. Les formules de cinq lettres nécessitent déjà de mémoriser $3,689 \times 10^{19}$ structures, ce qui n'est plus du tout réaliste. Mais nous pourrions envisager d'améliorer notre algorithme de simplification pour garantir une minimisation ou une quasi-minimisation de telles formules, ou de formules avec plus de lettres encore. Il serait souhaitable que ces formules réduites puissent jouer le rôle de forme canonique, ce qui les rendrait compétitives avec les BDDs dans les applications où la taille de ceux-ci "explose". Si un tel objectif est réalisable, il demandera très probablement d'ordonner les structures des collections de manière non triviale, ce qui compliquera substantiellement les algorithmes actuels des opérations *toSet* et *substitute*.

L'idée, au coeur des collections de structures, de représenter, de manière exhaustive, tous les termes égaux aux sous-termes d'un ensemble

(éventuellement infini) d'équations closes pourrait, peut-être, être généralisée à des ensembles de formules logiquement équivalentes utilisant comme atomes des égalités entre termes ; par exemple, des formules exprimant des propriétés arithmétiques. On pourrait alors envisager de décider l'équivalence de certaines formules en testant l'identité de formules minimisées.

Finalement, nous avons développé les collections de structures, leur implémentation, et leur validation expérimentale en vase clos, sans avoir eu le temps de les intégrer à des systèmes existants représentatifs de "l'état de l'art". Il serait souhaitable que nous développions une implémentation spécifique des collections de structures pour les intégrer, par exemple, aux outils de la librairie SMT-LIB [BFT16] et étudier dans quelle mesure elles permettent d'améliorer la résolution de certains problèmes. En particulier, nous avons comparé nos algorithmes aux algorithmes de fermeture congruente de la littérature sur base des structures de bas niveau de notre implémentation mais d'autres structures, peut-être plus légères car plus spécifiques, sont utilisées dans les outils de vérification de l'état de l'art. Des comparaisons expérimentales avec ces outils implémentés sont nécessaires pour établir si les collections de structures peuvent améliorer l'efficacité pratique de ces outils existants.

Bibliographie

[Ati12] Mêton Mêton Atindehou. Simplification des expressions régulières. Promoteur B. Le Charlier, september 2012.

[Ati14] Mêton Mêton Atindehou. Simplification des expressions : Rapport préliminaire et expressions booléennes. Promoteur B. Le Charlier, June 2014.

[AU92] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, Inc., New York, NY, USA, 1992.

[Bah04] RJ Bahnsen. Essential prime implicate tester. *IBM Tech Disclosure Bulletin*, 24(5), 2004.

[Ber34] BA Bernstein. A Set of Four Postulates for Boolean Algebra in Terms of the" Implicative" Operation. *Transactions of the American Mathematical Society*, 36(4) :876–884, 1934.

[BFR17] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 214–230. Springer, 2017.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BH71] JG Bredeson and PT Hultina. Generation of prime implicants by direct multiplication. *IEEE Transactions on Computers*, 20(4) :475–476, 1971.

[BHMSV84] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*, volume 2. Springer Science & Business Media, 1984.

[Bin56] Kurt Bing. On simplifying truth-functional formulas. *The Journal of Symbolic Logic*, 21(3) :253–254, 1956.

[Bis71] N. N. Biswas. Minimization of Boolean Functions. *IEEE Transactions on Computers*, C-20(8) :925–929, Aug 1971.

[Bis86] N.N. Biswas. Computer-Aided Minimization Procedure for Boolean Functions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 5(2) :303–304, April 1986.

[Bla38] A. Blake. *Canonical expressions in Boolean algebra*. University of Chicago, 1938.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation : Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Bra82] R. Brayton. Fast recursive boolean function manipulation. *Proc. International Symposium on Circuits and Systems*, pages 58–62, 1982. cited By (since 1996)6.

[Bro79] Frank Markham Brown. On a convenient division of labor in the generation of prime implicants. *Computers & Electrical Engineering*, 6(4) :267–271, 1979.

[Bro90] Frank Markham Brown. *Boolean reasoning : the logic of Boolean equations*. Springer Science & Business Media, 1990.

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691, 1986.

[Bry92] Randal E Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3) :293–318, 1992.

[Bub72] V Bubenik. Weighting method for the determination of the irredundant set of prime implicants. *Computers, IEEE Transactions on*, 100(12) :1449–1451, 1972.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on : http ://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[CM78] Ashok K Chandra and George Markowsky. On the number of prime implicants. *Discrete Mathematics*, 24(1) :7–11, 1978.

[CM92] Olivier Coudert and Jean Christophe Madre. Implicit and incremental computation of primes and essential primes of Boolean functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 36–39. IEEE Computer Society Press, 1992.

[CM94] Olivier Coudert and Jean-Christophe Madre. Une approche intentionnelle du calcul des implicants premiers et essentiels des fonctions booléennes. *RAIRO-Theoretical Informatics and Applications*, 28(2) :125–149, 1994.

[CMF93] Olivier Coudert, Jean Christophe Madre, and Henri Fraisse. A new viewpoint on two-level logic minimization. In *Design Automation, 1993. 30th Conference on*, pages 625–630. IEEE, 1993.

[Cou91] Olivier Coudert. *SIAM : une boîte à outils pour la preuve formelle de systèmes séquentiels*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 1991.

[Cou94] Olivier Coudert. Two-level logic minimization : an overview. *Integration, the VLSI journal*, 17(2) :97–140, 1994.

[Cou95] Olivier Coudert. Doing Two-Level Logic Minimization 100 Times Faster. In *SODA*, pages 112–121, 1995.

[Cou97] O. Coudert. Solving graph optimization problems with ZBDDs. In *Proceedings European Design and Test Conference. ED TC 97*, pages 224–228. IEEE, Mar 1997.

[CR01] Vinícius P Correia and André I Reis. Classifying n-input Boolean functions. *Proc. IWS*, 2001, 2001.

[DF59] B Dunham and R Fridshal. The problem of simplifying logical expressions. *Journal of Symbolic Logic*, pages 17–19, 1959.

[DFLBM13] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 46–52. IEEE, 2013.

[DGN+98] Anatoli Degtyarev, Yuri Gurevich, Paliath Narendran, Margus Veanes, and Andrei Voronkov. The Decidability of Simultaneous Rigid E-Unification with One Variable. In *RTA*, volume 1379 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 1998.

[Dic91] Jeremy Dick. An Introduction to Knuth-Bendix Completion. *Comput. J.*, 34 :2–15, 02 1991.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite Systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 243–320. MIT Press, Cambridge, MA, USA, 1990.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the Common Subexpression Problem. *J. ACM*, 27(4) :758–771, October 1980.

[Fea11] Paul Feautrier. *Simplification of Boolean affine formulas*. PhD thesis, INRIA, 2011.

[GF64] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5) :301–303, 1964.

[GH08] Steven Givant and Paul Halmos. *Introduction to Boolean algebras*. Springer Science & Business Media, 2008.

[Gha57] MJ Ghazala. Irredundant disjunctive and conjunctive forms of a Boolean function. *IBM Journal of Research and Development*, 1(2), 1957.

[GK99] Vladimir Gurvich and Leonid Khachiyan. On generating the irredundant conjunctive and disjunctive normal forms of monotone Boolean functions. *Discrete Applied Mathematics*, 96 :363–373, 1999.

[GMG84] CC Guest, MM Mirsalehi, and Thomas K Gaylord. Residue Number System Truth-Table Look-Up Processing ? Moduli Selection and Logical Minimization. *IEEE transactions on computers*, 33(10) :927–931, 1984.

[GN04] Ghoushchi MB Ghaznavi and AAR Nabavi. Simplification of Boolean Functions Using Boolean Differences. *SUMMER 2004*, 11(3) :203–217, 2004.

[Gra53] Frank Gray. Pulse code communication, March 17 1953. US Patent 2,632,058.

[GRS87] Jean H. Gallier, Stan Raatz, and Wayne Snyder. Theorem Proving Using Rigid E-Unification Equational Matings. In *LICS*, pages 338–346. IEEE Computer Society, 1987.

[HCO74] Se June Hong, Robert G. Cain, and Daniel L. Ostapko. MINI : A heuristic approach for logic minimization. *IBM journal of Research and Development*, 18(5) :443–458, 1974.

[HG78] Constantine Halatsis and Nicolaos Gaitanis. Irredundant normal forms and minimal dependence sets of a Boolean function. *IEEE Transactions on Computers*, C-27(11) :1064–1068, 1978.

[HS00] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 2000.

[Hun33] Edward V Huntington. New sets of independent postulates for the algebra of logic, with special reference to Whitehead and Russell's Principia Mathematica. *Transactions of the American Mathematical Society*, 35(1) :274–304, 1933.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In *Rewriting Techniques and Applications*, pages 23–37. Springer, 1997.

[Kar53] Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *American Institute of Electrical Engineers, Part I : Communication and Electronics, Transactions of the*, 72(5) :593–599, 1953.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y., 1970.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[Koz77] Dexter Kozen. Complexity of Finitely Presented Algebras. In *STOC*, pages 164–177. ACM, 1977.

[KT90] Alex Kean and George Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2) :185–206, 1990.

[LA15] Baudouin Le Charlier and Mêton Mêton Atindehou. A Method to Simplify Expressions : Intuition and Preliminary Experimental

Results. In *IWIL@LPAR*, volume 40 of *EPiC Series in Computing*, pages 37–51. EasyChair, 2015.

[LA16] Baudouin Le Charlier and Mêton Mêton Atindehou. A Data Structure to Handle Large Sets of Equal Terms. In *SCSS*, volume 39 of *EPiC Series in Computing*, pages 81–94. EasyChair, 2016.

[Le 14] Baudouin Le Charlier. Remarques sur la méthode de Karnaugh. Communication personnelle, 15 Mars 2014.

[Lev00] VS Levchenkov. Solution of equations in Boolean algebra. *Computational Mathematics and Modeling*, 11(2) :154–163, 2000.

[LRSC01] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. The MIT press, 2001.

[MB10] Frank Markham Brown. McColl and Minimization. *History and Philosophy of Logic*, 31(4) :337–348, 2010.

[McC56] Edward J McCluskey. Minimization of boolean functions*. *Bell system technical Journal*, 35(6) :1417–1444, 1956.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[Min93a] Shin-ichi Minato. Fast generation of prime-irredundant covers from binary decision diagrams. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 76(6) :967–973, 1993.

[Min93b] Shin-ichi Minato. Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 272–277, New York, NY, USA, 1993. ACM.

[Min01] Shin-ichi Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2) :156–170, May 2001.

[MJ60] Thomas H Mott Jr. Determination of the irredundant normal forms of a truth function by iterated consensus of the prime implicants. *Electronic Computers, IRE Transactions on*, EC-9(2) :245–252, June 1960.

[Mor67] E. Morreale. Partitioned List Algorithms for Prime Implicant Determination from Canonical Forms. *IEEE Transactions on Electronic Computers*, EC-16(5) :611–620, Oct 1967.

[MS72] A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (Swat 1972)*, SWAT '72, pages 125–129, Washington, DC, USA, 1972. IEEE Computer Society.

[MSBSV93] Patrick C McGeer, Jagesh V Sanghavi, Robert K Brayton, and AL Sangiovanni-Vicentelli. ESPRESSO-SIGNATURE : A new exact minimizer for logic functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(4) :432–440, 1993.

[Nec67] NN Necula. A numerical procedure for determination of the prime implicants of a Boolean function. *Electronic Computers, IEEE Transactions on*, EC-16(5) :687–689, Oct 1967.

[Nel55] Raymond J Nelson. Simplest normal truth functions. *The Journal of Symbolic Logic*, 20(2) :105–108, 1955.

[NM02] Tomoko Ninomiya and Masao Mukaidono. Independence of each axiom in a set of axioms and complete sets of axioms of Boolean algebra. In *Multiple-Valued Logic, 2002. ISMVL 2002. Proceedings 32nd IEEE International Symposium on*, pages 185–191. IEEE, 2002.

[NM03] Tomoko Ninomiya and Masao Mukaidono. Complete and independent sets of axioms of boolean algebra. In *Multiple-Valued Logic, 2003. Proceedings. 33rd International Symposium on*, pages 169–174. IEEE, 2003.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2) :245–257, October 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2) :356–364, 1980.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-Producing Congruence Closure. In Giesl J, editor, *Proc. 16th International Conference on Rewriting Techniques and Applications (RTA-2004), Nara, Japan*, number 3467 in LNCS. Springer, 2005.

[PR88] Sharon R Perkins and Tom Rhyne. An algorithm for identifying and selecting the primed implicants of a multiple-output Boolean function. *IEEE transactions on computer-aided design of integrated circuits and systems*, 7(11) :1215–1218, 1988.

[Qui52] Willard V Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, pages 521–531, 1952.

[Qui55] Willard V Quine. A way to simplify truth functions. *American mathematical monthly*, pages 627–631, 1955.

[Qui59] Willard V Quine. On cores and prime implicants of truth functions. *American Mathematical Monthly*, pages 755–760, 1959.

[Ree73] IS Reed. Boolean difference calculus and fault finding. *SIAM Journal on Applied Mathematics*, 24(1) :134–143, 1973.

[Reu75] Bernd Reusch. Generation of prime implicants from subfunctions and a unifying approach to the covering problem. *Computers, IEEE Transactions on*, 100(9) :924–930, 1975.

[RK62] Paul J Roth and RM Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(2) :227–238, 1962.

[Rom08] Steven Roman. *Lattices and ordered sets*. Springer Science & Business Media, 2008.

[RSV87] Richard L Rudell and Alberto Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions*

on Computer-Aided Design of Integrated Circuits and Systems, 6(5) :727–750, 1987.

[Rud86] Richard L Rudell. Multiple-valued logic minimization for PLA synthesis. Technical report, California Univ Berkeley Electronics Research Lab, 1986.

[Sam63] E Sampathkumar. A set of minimum number of postulates for a Boolean Algebra. *Proceedings Mathematical Sciences*, 57(4) :254–258, 1963.

[SC63] EW Samson and L Calabi. On the Theory of Boolean Formulas : Minimal Including Sums, I. *Journal of the Society for Industrial & Applied Mathematics*, 11(2) :212–223, 1963.

[Sch13] Stephan Schulz. System Description : E 1.8. In *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[SCL70] James R Slagle, Chin-Liang Chang, and Richard CT Lee. A new algorithm for generating prime implicants. *Computers, IEEE Transactions on*, 100(4) :304–310, 1970.

[Šed07] Miloš Šeda. Heuristic set-covering-based postprocessing for improving the Quine-McCluskey method. *International Journal of Computational Intelligence*, 4(2) :139–143, 2007.

[Sha38] C. E. Shannon. *A symbolic analysis of relay and switching circuits*, volume 57. IEEE, Dec 1938.

[Sho78] Robert E. Shostak. An Algorithm for Reasoning About Equality. *Commun. ACM*, 21(7) :583–585, July 1978.

[Sny93] Wayne Snyder. A Fast Algorithm for Generating Reduced Ground Rewriting Systems from a Set of Ground Equations. *J. Symb. Comput.*, 15(4) :415–450, 1993.

[SP12] Bernd Steinbach and Christian Posthoff. Improvements of the construction of exact minimal covers of boolean functions. In *Computer Aided Systems Theory–EUROCAST 2011*, pages 272–279. Springer, 2012.

[SRV01] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term Indexing. In *Handbook of Automated Reasoning*, pages 1853–1964, Amsterdam, The Netherlands, 2001.

[Sto36] Marshall H Stone. The theory of representation for Boolean algebras. *Transactions of the American Mathematical Society*, 40(1) :37–111, 1936.

[Sto15] A. Stoughton. *Formal Language Theory : Integrating Experimentation and Proof*. Cambridge University Press, 2015.

[Str92] Tadeusz Strzemecki. Polynomial-time algorithms for generation of prime implicants. *Journal of Complexity*, 8(1) :37–63, 1992.

[Tha81] André Thayse. Universal algorithms for evaluating boolean functions. *Discrete Applied Mathematics*, 3(1) :53–65, 1981.

[Tha04] André Thayse. *Calcul différentiel pour les langues de la logique : Théorie et applications*. Paris, Hermes - Lavoisier, april 2004.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic. *Mathematical Systems Theory*, 2(1) :57–81, 1968.

[UVSV06] Christopher Umans, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7) :1230–1246, 2006.

[Vaz13] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[Vid91] Lazare Georges Vidiani. Algèbre de Boole : axiomes minimaux. *Quadrature*, pages 13–15, 1991.

[Wec92] W. Wechler. *Universal algebra for computer scientists*. EATCS monographs on theoretical computer science. Springer-Verlag, 1992.

[Weg87] Ingo Wegener. *The complexity of Boolean functions*. John Wiley & Sons, Inc., 1987.

Table des figures

Liste des tableaux

Annexe A

Opérations sur les formules booléennes et stratégie "diviser pour régner"

C'est dans la "nature même" des formules booléennes de se prêter à la construction d'algorithmes appliquant la stratégie "diviser pour régner". Si F est une formule canonique, elle se décompose sous la forme $\overline{l}.F_0 \cup l.F_1$, et si c'est une formule quelconque (sous forme normale disjonctive), elle s'écrit sous la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$, où les formules F_0, F_1 et F_* ne contiennent pas la lettre l. On peut donc tenter de découper le problème à traiter concernant F, en deux ou trois sous-problèmes dont la somme des tailles est égale à celle du problème initial. Si n est le nombre de termes de F et m le nombre de lettres, on peut donc espérer résoudre le problème en un temps égal à $O(n \times m)$ car il y a en tout m niveaux d'appels et la somme des tailles des problèmes d'un même niveau est la même. En moyenne, on a $m = \log_2 n + 1$ [1]. Donc, on peut viser une complexité en $O(n \log n)$ pour ces algorithmes.

En pratique, malheureusement, les choses ne sont pas tout à fait aussi simples car, lorsqu'elle est possible, la découpe du problème est généralement moins directe. Dans cette annexe, nous montrons ce qu'il en est pour quatre opérations utilisées dans notre implémentation. (Nous avons déjà montré comment l'idée ci-dessus peut être adaptée pour l'algorithme de calcul des impliquants premiers de Coudert.)

Mise sous forme canonique d'une formule sous forme normale disjonctive Cette opération fournit un exemple (presque) idéal d'ap-

1. À strictement parler, c'est vrai seulement pour les formules canoniques.

plication de la stratégie “diviser pour régner”, expliquée ci-dessus.

Soit F, une formule utilisant m lettres. On construit sa forme canonique FC comme suit.

1. Si $m = 0$, on renvoie $FC = F$.
2. Sinon, F s'écrit sous la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$ où les formules F_0, F_1 et F_* ne contiennent pas la lettre l.

   On construit récursivement les formules canoniques FC_0 et FC_1 logiquement équivalentes aux formules $F_0 \cup F_*$ et $F_1 \cup F_*$, respectivement.

   On renvoie $FC = \overline{l}.FC_0 \cup l.FC_1$.

La correction de cet algorithme se prouve comme suit.

Preuve de correction.

1. Si $m = 0$, la formule F ne contient aucune lettre. Donc, ou bien $FC = 0$, ou bien $FC = 1$. Remarquons que $FC = 1$ n'est correct que parce que F ne contient pas de lettre.
2. Si $m > 0$, le résultat est correct parce que $\overline{l}.FC_0 \cup l.FC_1$ est sous forme canonique puisque les formules FC_0 et FC_1 le sont, par hypothèse d'induction, et sont donc formées de termes utilisant $m - 1$ lettres. De plus, les formules $\overline{l}.F_0 \cup l.F_1 \cup F_*$, $\overline{l}.(F_0 \cup F_*) \cup l.(F_1 \cup F_*)$, et $\overline{l}.FC_0 \cup l.FC_1$ sont logiquement équivalentes. □

Élimination des termes qui en impliquent un autre C'est l'opération que nous avons notée $\texttt{absorber}(F)$. Elle utilise comme sous-problème l'opération $\texttt{reduire}(F, G)$ qui retire de F les termes qui impliquent un terme de G.

L'algorithme de l'opération $\texttt{absorber}(F)$ est comme suit.

Soit F, une formule utilisant m lettres.

1. Si $m = 0$, F vaut 0 ou 1 et ne contient donc aucun terme qui en implique un autre. On renvoie F.
2. Sinon, F s'écrit sous la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$ où les formules F_0, F_1 et F_* ne contiennent pas la lettre l.

   On construit récursivement les formules A'_0, A'_1 et A_*, égales, respectivement, à $\texttt{absorber}(F_0)$, $\texttt{absorber}(F_1)$, et $\texttt{absorber}(F_*)$.

   On construit ensuite $A_0 = \texttt{reduire}(A'_0, A_*)$ et
   $A_1 = \texttt{reduire}(A'_1, A_*)$.

   On renvoie la formule $A = \overline{l}.A_0 \cup l.A_1 \cup A_*$.

Nous prouvons la correction de l'algorithme ci-dessus.

Preuve de correction. Le cas de base est "évident". Dans le cas général, on peut d'abord supposer, par hypothèse d'induction, que les formules A'_0, A'_1 et A_* contiennent exactement les termes de F_0, F_1 et F_* qui n'en impliquent aucun autre.

Donc les termes de A_0 et A_1 contiennent, respectivement, tous les termes de F_0 et F_1 qui n'impliquent aucun terme de F_0 ou de F_1 autre qu'eux même et qui n'en impliquent aucun de A_*, donc aucun de F_* (puisque, si c'était le cas, elle en impliquerait un de A_*).

Soit alors un terme de F. S'il est de la forme $l.t$ et qu'il implique un terme de F autre que lui-même, ou bien t implique un terme de F_*, alors t implique un terme de A_*, donc t n'appartient pas à A_1, et donc $l.t$ pas à $l.A_1$. Si ce terme $l.t$ n'implique aucun terme de F autre que lui-même, t n'implique aucun terme de F_*, donc aucun terme de A_*. De plus, t n'implique aucun terme de F_1 autre que lui-même et appartient donc à A'_1, et donc aussi à A_1 puisqu'il n'implique aucun terme de A_*. Bref $l.t$ appartient à $l.A_1$, donc à A.

On montre de la même façon qu'un terme de la forme $\overline{l}.t$ appartient à A s'il appartient à F et qu'il n'implique aucun autre terme de F.

Il reste le cas d'un terme t qui ne contient pas la lettre l. Il ne peut impliquer aucun terme de $\overline{l}.F_0$ ni de $l.F_1$. Il appartient donc à A si et seulement si il n'implique aucun terme de F_* différent de lui-même, c'est-à-dire si et seulement si il appartient à A_*, donc à A. □

Passons à l'algorithme de l'opération $\texttt{reduire}(F, G)$.

On suppose que la formule F utilise m lettres.

1. Si $m = 0$, ou bien $F = 0$ et le résultat est 0, ou bien $F = 1$ et le résultat est 1, sauf si G contient le terme 1, auquel cas, le résultat est 0.
2. Sinon on choisit une lettre l parmi celles qui sont utilisées par F. Les deux formules se décomposent respectivement sous la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$ et $\overline{l}.G_0 \cup l.G_1 \cup G_*$.
   On calcule récursivement les formules $B_0 = \texttt{reduire}(F_0, G_0 \cup G_*)$, $B_1 = \texttt{reduire}(F_1, G_1 \cup G_*)$, et $B_* = \texttt{reduire}(F_*, G_*)$.
   On renvoie alors la formule $B = \overline{l}.B_0 \cup l.B_1 \cup B_*$.

Preuve de correction. Il est à nouveau immédiat que le cas de base est correct. Considérons le cas général. Soit un terme appartenant à F.

Supposons d'abord qu'il est de la forme $l.t$. S'il implique un terme de G, t doit impliquer un terme de G_1 ou de G_* et t appartient à F_1. Donc, t n'appartient pas à B_1 et, par conséquent, $l.t$ n'appartient pas à B. Si

ce terme $l.t$ n'implique aucun terme de G, t n'implique aucun terme de $G_1 \cup G_*$ et appartient donc à B_1, puisqu'il appartient à F_1. Donc $l.t$ appartient à B.

Par un raisonnement semblable on montre qu'un terme de la forme $\overline{l}.t$ appartient à B si et seulement si il appartient à F et n'implique aucun terme de G.

Reste le cas d'un terme t appartenant à F et ne contenant pas la lettre l. Alors, il ne peut impliquer aucun terme de G, sauf ceux de G_*. Donc, il appartient à B_* et, par conséquent, à B si et seulement si il n'implique aucun terme de G. □

Remarque 3. Une optimisation de cet algorithme consiste à ne pas calculer les formules $G_0 \cup G_*$ et $G_1 \cup G_*$ et à les remplacer par 1, dès qu'elles contiennent le terme 1. Cette condition est immédiate à vérifier avec notre représentation des formules car le tableau représentant une formule a son premier élément égal à 0 si et seulement si la formule contient le terme 1.

Opération $\hat{.}$ Pour terminer, nous considérons le cas de l'opération $\hat{.}$. Un algorithme pour calculer $F\hat{.}G$ doit, dans le cas général, calculer trois formules : $C_0 = F_0\hat{.}G_0 \cup F_0\hat{.}G_* \cup F_*\hat{.}G_0$, $C_1 = F_1\hat{.}G_1 \cup F_1\hat{.}G_* \cup F_*\hat{.}G_1$, et $C_* = F_*\hat{.}G_*$, ce qui correspond à une décomposition du problème initial en sept sous-problèmes. Il se fait qu'on peut se ramener à une décomposition en trois sous-problèmes si on se "contente" de calculer une "approximation" de $F\hat{.}G$ que nous notons $\texttt{conj}(F, G)$, au moyen de l'algorithme ci-dessous.

Soient F et G, deux formules utilisant au plus m lettres (les mêmes).

1. Si $F = 0$ ou $G = 0$, on renvoie 0.
2. Si $F = 1$, on renvoie G. Symétriquement, si $G = 1$, on renvoie F.
3. Dans tous les autres cas, on choisit une lettre l, parmi les m lettres possibles. Les formules F et G se décomposent sous la forme $\overline{l}.F_0 \cup l.F_1 \cup F_*$ et $\overline{l}.G_0 \cup l.G_1 \cup G_*$, où les différentes sous-formules ne contiennent pas la lettre l.

   On calcule récursivement les formules $C_0 = \texttt{conj}(F_0 \cup F_*, G_0 \cup G_*)$, $C_1 = \texttt{conj}(F_1 \cup F_*, G_1 \cup G_*)$ et $C_* = \texttt{conj}(F_*, G_*)$.

   On renvoie enfin la formule $C = \overline{l}.(C_0 \setminus C_*) \cup l.(C_1 \setminus C_*) \cup C_*$.

Nous allons démontrer que $\texttt{conj}(F, G) = \texttt{absorber}(F\hat{.}G)$. L'opération `conj` nous suffit donc pour implémenter la méthode de Thayse, puisque son résultat doit de toute façon être simplifié par la suite.

Nous utilisons le lemme suivant :

Lemme 13. *Si A et B sont deux formules quelconques, on a :*

$$\texttt{absorber}(A \cup B) \setminus \texttt{absorber}(B) = \texttt{reduire}(\texttt{absorber}(A), \texttt{absorber}(B)).$$

Preuve du lemme 13. Nous prouvons l'inclusion dans les deux sens.

Supposons d'abord que t appartient à $\texttt{absorber}(A \cup B)$ mais pas à $\texttt{absorber}(B)$. Alors, t n'appartient pas à B. Car, si c'était le cas, ou bien il n'impliquerait aucun terme de B autre que lui-même, alors il appartiendrait à $\texttt{absorber}(B)$, ce qui est impossible, par hypothèse, ou bien il impliquerait un terme de B différent de lui-même, mais alors il n'appartiendrait pas à $\texttt{absorber}(A \cup B)$. Donc, t appartient à A et il n'implique aucun terme, ni de A, ni de B. Donc, il appartient à $\texttt{absorber}(A)$ mais pas à $\texttt{absorber}(B)$, donc il appartient à $\texttt{reduire}(\texttt{absorber}(A), \texttt{absorber}(B))$.

Dans l'autre sens, si t appartient à $\texttt{reduire}(\texttt{absorber}(A), \texttt{absorber}(B))$, il appartient à A et n'implique aucun terme de B ni aucun terme de A autre que lui-même. Donc, il appartient à $\texttt{absorber}(A \cup B)$. D'autre part, il n'implique aucun terme de $\texttt{absorber}(B)$, il ne peut donc pas appartenir à $\texttt{absorber}(B)$, puisqu'il s'implique lui-même. □

Preuve de correction (du fait que $\texttt{conj}(F, G) = \texttt{absorber}(F \barwedge G)$*).*

Les deux cas de base vont de soi si les formules sont simplifiées. Sinon, on peut renvoyer, selon le cas, $\texttt{absorber}(F)$ ou $\texttt{absorber}(G)$.

Dans le cas général, on montre successivement les propriétés suivantes.

a) $F \barwedge G = \overline{l}.(F_0 \barwedge G_0 \cup F_0 \barwedge G_* \cup F_* \barwedge G_0) \cup l.(F_1 \barwedge G_1 \cup F_1 \barwedge G_* \cup F_* \barwedge G_1) \cup F_* \barwedge G_*,$
par définition de l'opération $\barwedge$.

b) $\texttt{absorber}(F \barwedge G) = \overline{l}.\texttt{reduire}(\texttt{absorber}(F_0 \barwedge G_0 \cup F_0 \barwedge G_* \cup F_* \barwedge G_0), \texttt{absorber}(F_* \barwedge G_*)) \cup l.\texttt{reduire}(\texttt{absorber}(F_1 \barwedge G_1 \cup F_1 \barwedge G_* \cup F_* \barwedge G_1), \texttt{absorber}(F_* \barwedge G_*)) \cup \texttt{absorber}(F_* \barwedge G_*),$
en appliquant la propriété
$\texttt{absorber}(\overline{l}.F'_0 \cup l.F'_1 \cup F'_*) = \overline{l}.\texttt{reduire}(\texttt{absorber}(F'_0), \texttt{absorber}(F'_*)) \cup l.\texttt{reduire}(\texttt{absorber}(F'_1), \texttt{absorber}(F'_*)) \cup \texttt{absorber}(F'_*),$
démontrée dans la preuve de correction de l'opération absorber, à la formule du point 2.

c) $\texttt{absorber}(F \mathbin{\hat{\cdot}} G) =$
$\overline{l}.(\texttt{absorber}((F_0 \cup F_*) \mathbin{\hat{\cdot}} (G_0 \cup G_*)) \setminus \texttt{absorber}(F_* \mathbin{\hat{\cdot}} G_*)) \cup$
$l.(\texttt{absorber}((F_1 \cup F_*) \mathbin{\hat{\cdot}} (G_1 \cup G_*)) \setminus \texttt{absorber}(F_* \mathbin{\hat{\cdot}} G_*)) \cup$
$\texttt{absorber}(F_* \mathbin{\hat{\cdot}} G_*)$,
en appliquant le lemme démontré plus haut aux formules $(F_0 \cup F_*) \mathbin{\hat{\cdot}} (G_0 \cup G_*)$ et $F_* \mathbin{\hat{\cdot}} G_*$, ainsi qu'aux formules $(F_1 \cup F_*) \mathbin{\hat{\cdot}} (G_1 \cup G_*)$ et $F_* \mathbin{\hat{\cdot}} G_*$ et en remarquant que
$(F_0 \cup F_*) \mathbin{\hat{\cdot}} (G_0 \cup G_*) = (F_0 \mathbin{\hat{\cdot}} G_0 \cup F_0 \mathbin{\hat{\cdot}} G_* \cup F_* \mathbin{\hat{\cdot}} G_0) \cup F_* \mathbin{\hat{\cdot}} G_*$ et
$(F_1 \cup F_*) \mathbin{\hat{\cdot}} (G_1 \cup G_*) = (F_1 \mathbin{\hat{\cdot}} G_1 \cup F_1 \mathbin{\hat{\cdot}} G_* \cup F_* \mathbin{\hat{\cdot}} G_1) \cup F_* \mathbin{\hat{\cdot}} G_*$.

d) Donc, par hypothèse d'induction, l'algorithme renvoie la formule
$C = \overline{l}.(C_0 \setminus C_*) \cup \overline{l}.(C_1 \setminus C_*) \cup C_*$
$= \overline{l}.(\texttt{absorber}((F_0 \cup F_*) \mathbin{\hat{\cdot}} (G_0 \cup G_*)) \setminus \texttt{absorber}(F_* \mathbin{\hat{\cdot}} G_*)) \cup$
$l.(\texttt{absorber}((F_1 \cup F_*) \mathbin{\hat{\cdot}} (G_1 \cup G_*)) \setminus \texttt{absorber}(F_* \mathbin{\hat{\cdot}} G_*)) \cup$
$\texttt{absorber}(F_* \mathbin{\hat{\cdot}} G_*)$,
$= \texttt{absorber}(F \mathbin{\hat{\cdot}} G)$

□

Annexe B

Deux théorèmes utilisés par l'étude de complexité des collections de structures

Nous démontrons d'abord une propriété utilisée dans la preuve du théorème 23.

Théorème 32. *Soient a et b deux nombres strictement positifs. Si $a \leq b$, on a :*

$$a + \frac{1}{2}(a \log_2 a + b \log_2 b) \;\leq\; \frac{1}{2}(a+b)\log_2(a+b) \tag{B.1}$$

Preuve du théorème 32. Il suffit de démontrer que la fonction de la variable réelle x définie par l'expression

$$\frac{1}{2}(a+x)\log_2(a+x) - a - \frac{1}{2}(a \log_2 a + x \log_2 x) \tag{B.2}$$

est positive pour toute valeur de x supérieure ou égale à a. On vérifie d'abord facilement que la valeur de l'expression B.2 est nulle lorsque $x = a$. Nous montrons ensuite que la fonction est croissante lorsque x prend des valeurs supérieures ou égales à a. Ces conditions impliquent le résultat. On voit aisément que la fonction est croissante si la fonction définie par l'expression plus simple ci-dessous l'est :

$$(a+x)\log_2(a+x) - x \log_2 x \tag{B.3}$$

La dérivée de cette fonction est donnée par l'expression :

$$\log_2(a+x) - \log_2 x \tag{B.4}$$

qui est positive pour toute valeur strictement positive de x. □

Théorème 33. *Soient $a_1, \ldots, a_q$, des nombres tels que $a_i \geq 1$ $(i = 1, \ldots, q)$ et soumis à la contrainte $a = a_1 + \ldots + a_q$ $(a \geq q)$. (Donc, a est fixé mais les a_i "varient librement" sous cette contrainte.)*
Dans ces conditions, l'expression

$$\sum_{i=1}^{q} a_i \log a_i \tag{B.5}$$

possède un minimum égal à

$$a \log \frac{a}{q}. \tag{B.6}$$

Ce minimum est atteint lorsque

$$a_1 = \ldots = a_q = \frac{a}{q}. \tag{B.7}$$

Preuve du théorème 33. Remarquons d'abord, que la véracité du théorème ne dépend pas de la base choisie pour le logarithme puisque des logarithmes calculés dans deux bases différentes ne diffèrent que par un facteur constant. Nous utilisons donc des logarithmes népériens (en base e) pour la simplicité.
Nous démontrons le théorème par récurrence sur la valeur de q. L'énoncé suppose implicitement que $q \geq 1$. Le cas de base est donc $q = 1$ et le théorème est vrai, dans ce cas, puisque l'expression B.5 ne peut prendre que la valeur $a \log a$.
Supposons donc que $q > 1$ et "fixons momentanément" la valeur a_1. Les hypothèses du théorème sont vérifiées si on substitue les nombres $q - 1$, $a - a_1$ et $a_2, \ldots, a_q$ aux nombres q, a et $a_1, \ldots, a_q$, de l'énoncé. On peut, dès lors, supposer, par hypothèse de récurrence sur q, que

$$\sum_{i=2}^{q} a_i \log a_i \geq (a - a_1) \log \frac{a - a_1}{q - 1}. \tag{B.8}$$

On a, par conséquent, aussi :

$$\sum_{i=1}^{q} a_i \log a_i \geq a_1 \log a_1 + (a - a_1) \log \frac{a - a_1}{q - 1}. \tag{B.9}$$

Considérons maintenant la fonction définie, pour x tel que $1 \leq x \leq a - q + 1$, par l'expression

$$x \log x + (a - x) \log \frac{a - x}{q - 1}. \tag{B.10}$$

Nous allons montrer plus bas que cette fonction possède un minimum égal à

$$a \log \frac{a}{q}. \tag{B.11}$$

Il s'en suit, puisque $1 \leq a_1 \leq a - q + 1$, que la relation B.9 peut se simplifier en

$$\sum_{i=1}^{q} a_i \log a_i \ \geq \ a \log \frac{a}{q}. \tag{B.12}$$

Cette relation suffit à établir le théorème, en tenant compte du fait que

$$\sum_{i=1}^{q} a_i \log a_i \ = \ a \log \frac{a}{q}$$

lorsque

$$a_1 \ = \ \ldots \ = \ a_q \ = \ \frac{a}{q}.$$

(Notons qu'on a bien aussi $\frac{a}{q} \geq 1$.)

Il nous reste à prouver l'affirmation B.11. La dérivée de l'expression B.10 est égale à

$$\log(q-1) \ - \ \log \frac{a-x}{x}. \tag{B.13}$$

Elle s'annule lorsque

$$x = \frac{a}{q}. \tag{B.14}$$

On vérifie que la valeur de l'expression B.13 est négative lorsque $1 \leq x < \frac{a}{q}$ et qu'elle est positive si $\frac{a}{q} < x$. La valeur $x = \frac{a}{q}$ correspond donc bien à un minimum de l'expression B.10. On vérifie aisément qu'elle donne à cette expression la valeur $a \log \frac{a}{q}$, comme annoncé.

□

Annexe C

Compléments sur le rapport entre estimations et valeurs expérimentales du nombre de renommages

Cette annexe contient les graphiques des courbes illustrant la discussion de la section 2.5.4. Les courbes rouges, vertes, jaunes et bleues correspondent respectivement aux valeurs 1.000, 10.000, 100.000 et 1.000.000 pour le nombre de structures N.

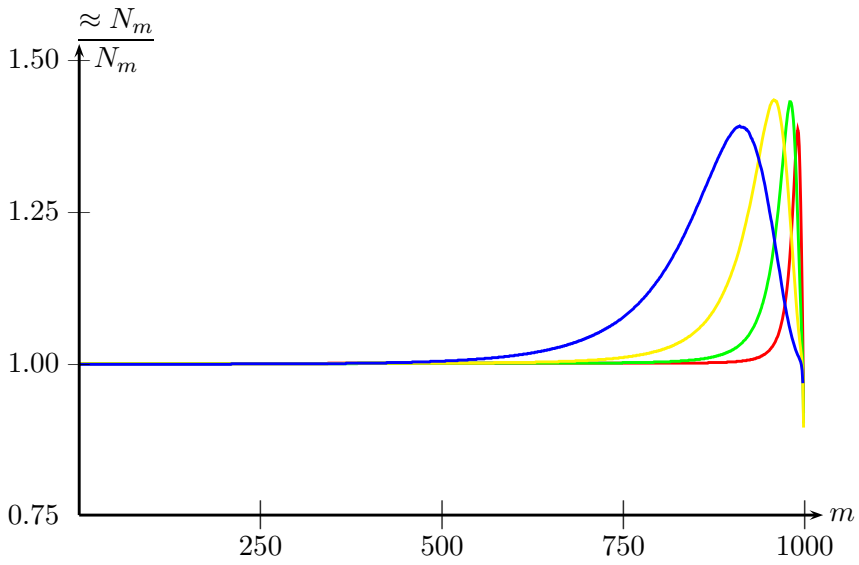


Figure C.1 – Rapport entre estimation et valeur expérimentale de N_m pour *substitute*

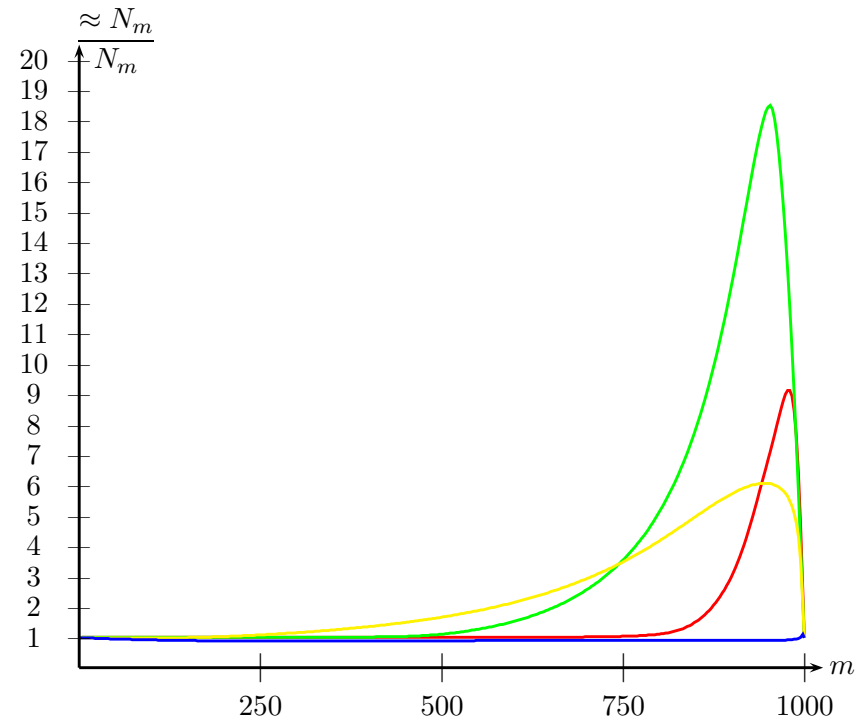


Figure C.2 – Rapport entre estimation et valeur expérimentale de N_m pour *unify*

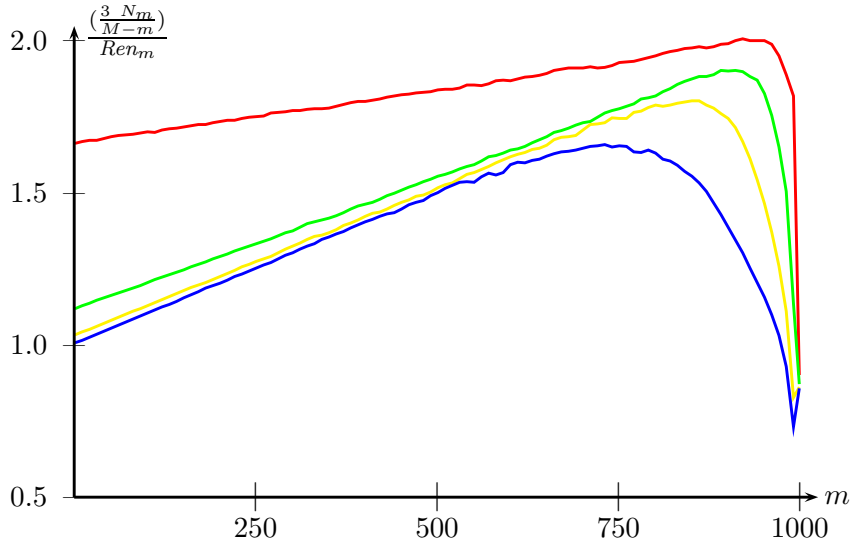


Figure C.3 – Rapport entre $\frac{3N_m}{(M-m)}$ et Ren_m pour *substitute*

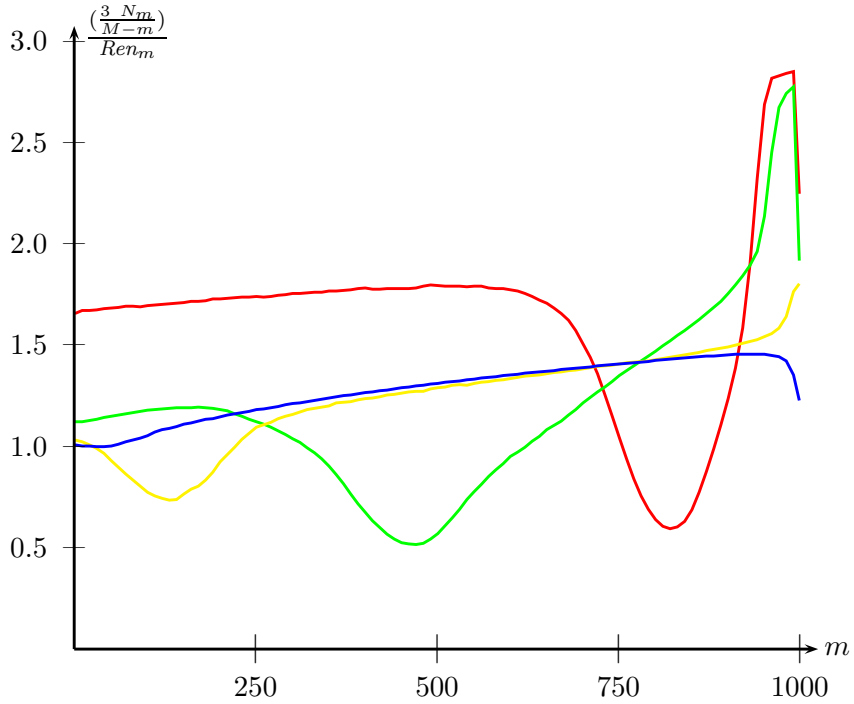


Figure C.4 – Rapport entre $\frac{3N_m}{(M-m)}$ et Ren_m pour *unify*