UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

par
Patrice Guérin

Génération des classes d'isomorphisme des boucles d'ordre 8

14 mai 2003

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Résumé

Les boucles sont des objets mathématiques qui sont étudiés depuis le début du siècle. Leur non-associativité les rend cependant difficiles à comprendre et leur nombre empêche l'étude exhaustive.

Plusieurs chercheurs abordent les boucles en les classifiant en classe d'isotopie. Toutefois, ces classes n'ont pas un sens propre si on utilise les boucles pour la reconnaissance de langages. Il faut plutôt, dans ce cas, s'intéresser aux classes d'isomorphisme.

La génération des classes d'isomorphisme, même pour des ordres aussi petits que 8 est très difficile. Cette recherche présente des algorithmes permettant de travailler la génération des classes d'isomorphisme de boucles en temps raisonnable pour les ordres inférieurs à 8.

Ces algorithmes peuvent également être utilisés comme algorithme de recherche pour des ordres supérieurs.

Remerciements

Je tiens d'abord à remercier mon directeur, M. François Lemieux, pour son support constant. Son soutien m'a permis de terminer cette recherche et surtout ce mémoire. Ses mots sur l'importance de notre travail pour la postérité ont donné un sens aux efforts quotidiens.

Également Marie-Hélène, ma conjointe, pour sa patience et ses encouragements. Son support moral a été essentiel pour la réalisation et la rédaction de ce projet.

Mes parents m'ont aussi supporté et aidé grandement. Ils m'ont poussé à reprendre l'ouvrage laissé sur le métier et incité à terminer la rédaction de cette étude. Merci tout spécialement à ma mère qui a bien voulu prendre de son temps et mettre la touche finale pour les corrections.

Un gros merci aux gens du Groupe de recherche en informatique (GRI) et plus particulièrement à Pierre que j'ai côtoyé 18 mois durant. Enfin un petit mot pour Renaud qui m'a donné du temps, ce bien si précieux.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

CHAPITRE 1

INTRODUCTION

Ce travail porte sur la génération de carrés latins et de boucles de taille finie. Cette génération est très complexe même pour de très petites boucles (moins de 10 éléments).

Les carrés latins sont des tableaux carrées de taille n dont les cases sont remplies par les éléments d'un ensemble de n éléments avec la restriction que deux éléments ne se retrouvent pas deux fois sur la même ligne ni sur la même colonne. Les carrés latins ont été étudiés au début par Euler qui utilisait les alphabets grecs et latins comme support, et c'est d'ailleurs de là que vient le nom carré latin.

Les boucles, quant à elles, sont des structures algébriques non associatives qui sont vraiment nées au début du siècle par les travaux de Moufang. Il existe un lien étroit entre les boucles et les carrés latins puisque les tables de multiplication des boucles finies sont des carrés latins.

Les boucles et les carrés latins ont fait l'objet d'études, surtout durant le dernier siècle, dans des domaines aussi variés que l'algèbre, la géométrie, la topologie et la combinatoire (voir [15] et [23]). Mais encore, de récentes études ouvrent d'autres domaines pour l'application des boucles. Elles sont en effet liées à la reconnaissance de langages formels (voir [1], [2] et [16]).

En effet, les monoïdes sont des structures algébriques associatives ayant un élément neutre. Nous savons déjà que les monoïdes permettent de reconnaître exactement les langages réguliers (voir [25]) et que les groupoïdes, structures algébriques sans restriction, reconnaissent exactement les langages hors-contextes (voir [3]). De plus, le théorème des variétés d'Eilenberg permet d'associer de façon bijective des familles de langages réguliers à des familles de monoïdes (voir [7]).

Aucune généralisation de ce théorème n'est connue pour les langages hors-contextes et les groupoïdes. Nous commençons donc par étudier les boucles pour arriver un jour à comprendre les groupoïdes.

Pour arriver à mieux connaître les boucles, nous pouvons générer toutes les boucles

et ensuite chercher des corrélations entre elles. Le nombre de boucles est extrêmement grand, même pour des ordres inférieurs à 10 (voir section 2.7). Nous utilisons donc la notion d'isomorphisme pour réduire le nombre de boucles à générer. En effet, deux boucles isomorphes reconnaissent exactement les mêmes langages. Dans le cas de la reconnaissance de langages, deux boucles isomorphes sont donc équivalentes. Un ensemble de boucles isomorphes entre elles est appelé classe d'isomorphisme. Notre projet consiste à générer un élément de chaque classe d'isomorphisme de boucles pour les ordres inférieurs à 9.

Pour $n = 7$, un algorithme prenant quelques heures sur une machine cadencée à 500 MHz était connu. Cependant, ce même algorithme prendrait quelques 15 ans à répondre pour $n = 8$. Notre objectif a donc été d'analyser théoriquement les boucles et surtout les représentants de boucles afin d'arriver à générer les classes d'isomorphisme d'ordre 8 en temps raisonnable. Nous croyons avoir réduit la complexité de cette génération dans l'ordre du nombre de classes d'isomorphisme de boucles plutôt que dans l'ordre du nombre de boucles. L'incertitude naît du fait que notre intuition n'est appuyée que par des résultats empiriques, l'analyse théorique dépassant le cadre de ce mémoire.

Ce mémoire présente les résultats de notre recherche pour arriver à générer un élément de chaque classe d'isomorphisme de boucles d'ordre 8. Ce résultat original a été atteint au mois de juin 2001. Nous avons découvert des choses intéressantes sur les représentants de classes d'isomorphisme de boucles (voir chapitres 4 et 5). Les chapitres 2 et 3 sont une introduction aux boucles et une présentation du vocabulaire propre à notre recherche.

Le chapitre 2 se divise en deux parties, l'une explique la nature des boucles et l'autre, les langages formels. La toute première section fournit les définitions générales sur les ensembles et les groupoïdes. Celles-ci seront ensuite utilisées pour le reste de

cette étude. Ensuite, les classes d'isomorphisme et d'isotopie sont présentées. C'est l'objet principal de notre recherche puisque nous voulons générer les boucles distinctes à un isomorphisme près. Les carrés latins, tables de multiplication de boucles finies, sont ensuite définis. Pour générer les boucles, nous générons en fait leur table de multiplication, en d'autres mots des carrés latins. Cette section contient un théorème portant sur les rectangles latins (voir [10]) qui sert à simplifier la représentation des algorithmes au chapitre 4.

La deuxième partie de ce chapitre commence par les définitions de base des langages formels. Les langages réguliers et hors-contextes sont ensuite définis. Cette section sert à comprendre pourquoi nous avons généré les classes d'isomorphisme de boucles plutôt que les boucles ou les classes d'isotopie de boucles. Trois types d'automates sont présentés sur le même thème. Il s'agit de l'automate fini déterministe, l'automate à pile déterministe et l'automate à pile non déterministe. Les liens avec les langages formels sont donnés par les familles de langages reconnues par ces différents automates. Finalement, la reconnaissance de langages par des monoïdes, des groupoïdes et des boucles est expliquée. En conclusion, la présentation des motivations de cette étude est faite.

Le chapitre 3 fournit la plupart des outils théoriques qui seront utilisés aux chapitres 4 et 5 pour générer les classes d'isomorphisme de boucles. Nous y parlons d'abord des permutations, puisque les lignes et les colonnes d'un carré latin en sont. Ensuite, nous définissons la notion de relation d'ordre qui nous permettra ensuite de comparer les boucles entre elles. Les classes d'isomorphisme sont ensuite expliquées de même que l'idée de représentants qui est le plus petit membre d'une classe d'isomorphisme de boucles. Ce sont ces représentants que les algorithmes que nous avons développés génèrent. Finalement, une étude des deuxièmes lignes de représentants de boucles les caractérisent exactement. Cette caractérisation des deuxièmes lignes des représentants

de classes d'isomorphisme de boucles est un des résultats importants de notre étude.

Le coeur de cette recherche se retrouve dans le chapitre 4. Il s'agit de la présentation de l'ensemble des algorithmes que nous avons développés pour générer plus rapidement les classes d'isomorphisme de boucles. Ce chapitre présente progressivement des algorithmes de plus en plus complexes et de plus en plus rapides. D'abord un léger survol des types d'algorithmes combinatoires est fait. Les algorithmes de cette recherche sont catégorisés lorsqu'ils sont présentés. Ensuite, un algorithme de base pour la génération des classes d'isomorphisme est présenté. Cet algorithme de base est lent et n'est viable que pour des ordres de 7 et moins.

La présentation des accélérations que nous avons trouvées occupe la seconde moitié de ce chapitre. Au nombre de trois, les deux premières restreignent la recherche de représentants à certains espaces de solutions. Il s'agit de l'accélération par les cycles et de l'accélération par les rectangles latins minimaux. La seconde accélération, la réduction des permutations, permet de fouiller plus rapidement l'espace solution. Ces accélérations forment le coeur de notre travail.

Bien que les algorithmes déjà présentés soient relativement aisés à comprendre, leur implémentation peut grandement influencer le temps d'exécution réel. Ainsi le chapitre 5 fournit les accélérations que nous avons découvertes grâce à une recherche empirique. Il s'agit d'abord du calcul de la loi d'annulation et celui des cycles qui sont redondants. Ensuite, des relations d'ordre différentes sont présentées.

Comme l'analyse du temps d'exécution théorique de notre algorithme est très difficile, nous fournissons une analyse basée sur des mesures trouvées empiriquement. Ainsi, une analyse des résultats quant aux accélérations gagnées par la réduction des permutations est donnée. Ensuite, une analyse empirique du temps réel d'exécution est expliquée. Nous croyons ce temps d'exécution dépendant du nombre de classes d'isomorphisme et nous appuyons notre intuition par des résultats expérimentaux. En

conclusion, une approximation du temps que prendrait une exécution pour $n = 9$ est donnée.

Dans le chapitre 6, nous présentons les utilisations futures de notre algorithme. Que ce soit pour l'étude des boucles d'ordres supérieurs à 8 ou pour la génération des classes d'isotopie. Enfin, les modifications qui pourraient permettre l'accélération de notre algorithme sont fournies. Certaines ne sont que des impressions, d'autres sont des certitudes que nous n'avons pas implantées.

CHAPITRE 2

GROUPOÏDES, QUASIGROUPES ET BOUCLES

Ce chapitre explique les concepts de base qui permettent de comprendre la présente recherche et son dessein. Nous commençons par les définitions essentielles sur les ensembles. Ensuite, nous parlons des carrés latins et des rectangles latins que nous utilisons au chapitre 4. Les deux sections suivantes présentent quelques langages formels et quelques automates pour saisir l'approche de notre recherche. L'avant dernière section introduit la reconnaissance de langages par des groupoïdes. En guise de conclusion, la dernière section introduit la présente recherche et sa motivation.

Nous commençons par définir les catégories de groupoïdes et leur hiérarchie à la section 2.1. Nous y voyons les définitions et propriétés qui permettent de nommer ces groupoïdes de même que leurs hiérarchies. Nous définissons également les concepts d'isomorphisme et d'isotopie importants pour notre étude.

La section 2.3 définit les carrés latins et les rectangles latins. Nous y voyons l'un des résultats les plus importants de notre étude, soit qu'il est toujours possible de compléter un rectangle latin pour obtenir un carré latin.

Ensuite, nous présentons les grandes lignes permettant de définir les langages formels à la section 2.4. Plus particulièrement, nous étudions les langages réguliers et hors-contextes qui sont les deux familles de langages les plus connus. Pour comprendre cette recherche, la section 2.5 décrit le modèle des automates qui est intimement lié à celui des langages formels. Nous parlons des automates finis déterministes et des automates à pile non déterministes. Les liens existants entre ces automates et les langages qu'ils reconnaissent sont décrits après la présentation de chaque automate.

Finalement, à la section 2.6, nous voyons que les groupoïdes peuvent être utilisés pour reconnaître des langages. Nous voyons que les monoïdes reconnaissent certaines familles de langages et les groupoïdes d'autres. Plus intéressant encore, les boucles reconnaissent exactement les langages réguliers ouverts. Finalement, nous expliquons le concept fondamental d'isomorphisme qui permet de comprendre la génération des

classes d'isomorphisme au chapitre 4.

En conclusion, la section 2.7 expose la complexité du problème qui nous intéresse : la génération de carrés latins. Nous verrons, de plus, pourquoi nous générerons les classes d'isomorphisme. Nous aurons également une idée de la complexité de ce problème par rapport à d'autres similaires, générer les groupes par exemple.

2.1 Définitions

Nous allons commencer par définir les notions de base et le vocabulaire qui sont utilisés ensuite tout au long de ce document. Le lecteur peut se référer à [24] pour plus de détails sur ces concepts, la notation seule est différente. Rappelons d'abord qu'un ensemble est une collection d'objets distincts, appelés éléments.

Définition 2.1 *Si A est un ensemble non vide, alors une fonction $\Phi : A \times A \rightarrow A$ est appelée une* opération binaire.

Exemple

Par exemple, on peut définir $+$ l'addition modulo 5 sur l'ensemble $\{0, 1, 2, 3, 4\}$. Ainsi on notera $2 + 4 = 1$.

Définition 2.2 *Un* groupoïde *est un ensemble muni d'une opération binaire. L'ensemble est appelé* support.

On note $(G, \cdot)$ le groupoïde de l'ensemble G et de l'opération $\cdot$. Lorsqu'il n'y a aucune ambiguïté, on note G le groupoïde $(G, \cdot)$ et l'opération est alors la concaténation. On remarque qu'il n'y a pas de restrictions à l'opération $\cdot$. On pourrait cependant décider de le faire. Ces restrictions sont appelées propriétés. La définition suivante nomme les principales propriétés que nous utilisons.

Définition 2.3 *Soit un groupoïde* $(G, \cdot)$,

$\forall a, b, c \in G, a \cdot b = a \cdot c \Rightarrow b = c$	Annulation à gauche
$\forall a, b, c \in G, b \cdot a = c \cdot a \Rightarrow b = c$	Annulation à droite
$\forall a, b, c \in G, a \cdot (b \cdot c) = (a \cdot b) \cdot c$	Associativité
$\forall a, b \in G, a \cdot b = b \cdot a$	Commutativité
$\exists e \in G, a \cdot e = e \cdot a = e$	*Existence d'un* élément neutre

Exemple

Reprenons l'ensemble $A = \{0, 1, 2, 3, 4\}$ *et* $+$ *l'addition modulo* 5. *On peut vérifier que toutes ces propriétés sont vérifiées dans le groupoïde* $(A, +)$. *On peut représenter un groupoïde à l'aide de sa table de multiplication. Ce qui donne dans le cas présent :*

$(A, +) =$

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Selon les propriétés associées à un groupoïde, nous donnons un nom plus précis à celui-ci. Les cinq définitions suivantes fournissent ce vocabulaire.

Définition 2.4 *Hiérarchie des groupoïdes*

1. *Un* semigroupe *est un groupoïde associatif.*
2. *Un* quasigroupe *est un groupoïde supportant les lois d'annulation à gauche et à droite.*

3. *Un* monoïde *est un semigroupe possédant un élément neutre.*

4. *Une* boucle *est un quasigroupe possédant un élément neutre.*

5. *Un* groupe *est une boucle associative.*

Il existe, comme on le voit, une hiérarchie bien déterminée des groupoïdes. La figure 1 représente cette hiérarchie.

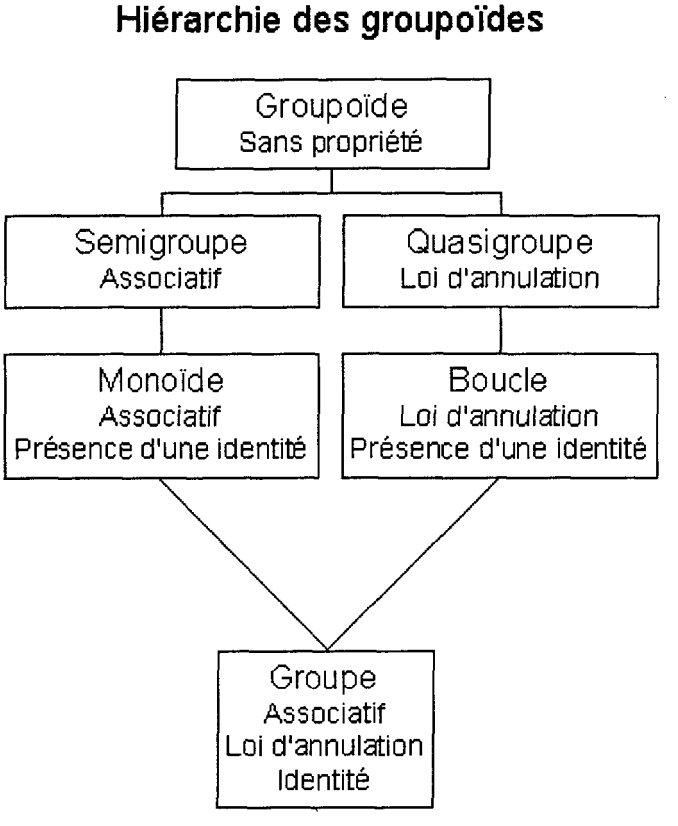

FIG. 1: Hiérarchie des groupoïdes

Les structures associatives ont été largement étudiées dans le passé. Il existe ainsi une théorie élaborée sur les semigroupes et les monoïdes. La situation des groupoïdes est différente. On a longtemps pensé que l'étude de ces objets n'étaient que des généralisations des résultats des structures associatives. Des résultats récents, se trouverant

dans [2] et [16], semblent montrer que les boucles particulièrement forment un domaine d'étude propre et riche.

En effet, la théorie des boucles est utilisée dans des domaines aussi variés que l'algèbre, la géométrie, la topologie et la combinatoire. Pour une courte historique de l'étude des boucles et des quasigroupes le lecteur peut se référer à [23].

2.1.1 L'évaluation d'une expression non associative

Il faut comprendre que si une opération n'est pas associative, alors la façon de parenthétiser cette expression change sa valeur. Parfois, il est intéressant de connaître toutes ces valeurs. C'est ce qu'introduit la définition suivante.

Définition 2.5 *Soit un groupoïde G et ω une expression quelconque sur G. Alors, $eval(\omega)$ est l'ensemble des valeurs possibles de l'expression ω dans G.*

Exemple

Prenons le groupoïde suivant :

$$(G,+) = \begin{array}{c|ccccc} + & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 2 & 3 & 4 \\ 1 & 1 & 0 & 3 & 4 & 2 \\ 2 & 2 & 3 & 4 & 1 & 0 \\ 3 & 3 & 4 & 0 & 2 & 1 \\ 4 & 4 & 2 & 1 & 0 & 3 \end{array}$$

Considérons l'expression $\omega = 2+2+2$. Alors il existe deux façons d'évaluer ω.

1. $(2+2)+2 = 4+2 = 1$
2. $2+(2+2) = 2+4 = 0$

Donc, $eval(\omega) = \{0,1\}$.

2.2 Les classes d'isomorphisme

Une notion assez naturelle des groupoïdes est celle d'isomorphisme. Il semble assez naturel que les deux groupoïdes suivants soient intimement liés.

$$(G,\cdot) = \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 1 & 0 & 1 \\ 0 & 1 & 0 \end{array} \qquad (H,\circ) = \begin{array}{c|cc} \circ & a & b \\ \hline a & a & b \\ b & b & a \end{array}$$

En effet, il est possible de renommer les éléments de G par ceux de H et ainsi la table de multiplication de H serait obtenue à partir de celle de G. Il suffit de renommer 0 par a et 1 par b. On dit que G et H sont isomorphes. Cette notion peut s'étendre à une autre plus générale appelée isotopie.

2.2.1 Les isomorphismes

Définition 2.6 *Deux groupoïdes $(G,\cdot)$ et $(H,+)$ sont* homomorphes *s'il existe une fonction $\alpha : G \longrightarrow H$, telle que :*

$$\alpha(x \cdot y) = \alpha(x) + \alpha(y).$$

Exemple

Les deux groupoïdes suivants, représentés par leur table de multiplication, sont homomorphes :

$$(G,\cdot) = \begin{array}{c|cccc} \cdot & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 1 & 0 & 3 & 2 \\ 2 & 2 & 3 & 0 & 1 \\ 3 & 3 & 2 & 1 & 0 \end{array} \qquad (H,+) = \begin{array}{c|cc} + & a & b \\ \hline a & a & b \\ b & b & a \end{array}$$

En effet, si on prend $\alpha : G \longrightarrow H$, tel que $\alpha(0) = a$, $\alpha(1) = a$, $\alpha(2) = b$, $\alpha(3) = b$, on voit que α répond à la définition 2.6.

Or, il peut arriver que G et H aient la même taille *i.e.* : $|G| = |H|$, et que α soit alors une fonction injective. On dit dans ce cas que α est un *isomorphisme.* Si les ensembles G et H sont identiques, on dit que α est un *renommage* des éléments de G. Pour le reste de ce document, à moins de mention contraire, c'est de ce type d'isomorphisme dont nous parlons. Nous notons $G \sim H$ pour exprimer G isomorphe à H.

Exemple

Soit les trois groupoïdes suivants :

$\cdot$	0	1	2	3	4
0	0	1	2	3	4
1	1	0	3	4	2
2	2	3	4	1	0
3	3	4	0	2	1
4	4	2	1	0	3

Groupoïde $(A, \cdot)$

$\times$	0	1	3	4	2
0	0	1	3	4	2
1	1	0	4	2	3
3	3	4	2	1	0
4	4	2	0	3	1
2	2	3	1	0	4

Groupoïde $(A, \times)$

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	0	3	4	2
2	2	3	4	1	0
3	3	4	0	2	1
4	4	2	1	0	3

Groupoïde $(A, +)$

Ces groupoïdes sont isomorphes puisqu'il suffit de renommer les éléments du groupoïde $(A, \cdot)$ *pour arriver au groupoïde* $(A, \times)$. *L'isomorphisme* $h : A \rightarrow A = \{ (0,0), (1,1), (2,3), (3,4), (4,2) \}$*peut être utilisée par exemple. Pour des raisons de clarté et puisque cela ne change rien au sens des tables de multiplication, on replace les lignes et les colonnes de la table de multiplication* $(A, \times)$ *pour obtenir celle de* $(A, +)$. *En fait,* $(A, \times)$ *et* $(A, +)$ *sont identiques.*

Définition 2.7 *Soit un groupoïde* $(G, \cdot)$, *un* automorphisme *est un isomorphisme du groupoïde dans lui-même.*

Donc, si une permutation appliquée à un groupoïde préservent son opération, on dira que cette permutation est un automorphisme.

2.2.2 Les isotopies

Les isotopies forment une généralisation des isomorphismes. Plusieurs travaux ont été faits sur les isotopies de boucles plutôt que sur les isomorphismes. Nous allons, à la section 2.6.4, clarifier les raisons qui nous ont poussé à ne pas faire de même.

Définition 2.8 *Deux groupoïdes $(G,\cdot)$ et $(H,+)$ sont isotopiques s'il existe trois fonctions $\alpha : G \to H$, $\beta : G \to H$, $\gamma : G \to H$, de telle façon que $\alpha(a \cdot b) = \beta(a) + \gamma(b)$.*

On remarque immédiatement que l'isomorphisme est le cas particulier de l'isotopie où les trois fonctions sont identiques. En tel cas, $\alpha(a) = \beta(a) = \gamma(a)$ pour tout a.

Clarifions un peu l'interprétation à faire des isotopies. Prenons deux groupoïdes $(G,\cdot)$ et $(G,+)$ et représentons-les par leur table de multiplication. Alors $(G,\cdot)$ et $(G,+)$ sont isotopiques s'il existe une façon de permuter les symboles (la fonction α), les colonnes (la fonction β) et les lignes (la fonction γ) de la table de multiplication de $(G,\cdot)$ et d'obtenir la table de multiplication de $(G,+)$.

Exemple

Prenons les deux groupoïdes suivants :

$(G,\cdot) =$

$\cdot$	0	1	2	3	4
0	0	1	2	3	4
1	1	0	3	4	2
2	2	4	0	1	3
3	3	2	4	0	1
4	4	3	1	2	0

$(G,+) =$

$+$	1	0	2	3	4
0	1	0	2	3	4
1	0	1	3	4	2
4	2	4	1	0	3
2	3	2	4	1	0
3	4	3	0	2	1

ou

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	0	3	4	2
2	2	3	4	1	0
3	3	4	0	2	1
4	4	2	1	0	3

En utilisant $\alpha = (1,0,2,3,4)$, $\beta = (1,0,2,3,4)$ et $\gamma = (0,1,4,2,3)$, on vérifie que $\forall\ a,b \in G$, $\alpha(a \cdot b) = \beta(a) + \gamma(b)$. Donc les groupoïdes G et H sont isotopiques.

2.3 Les carrés latins

Le domaine combinatoire de l'étude des boucles est l'étude des carrés latins. Les tables de multiplication des boucles forment des carrés latins. Comme nous allons représenter les boucles dans un algorithme à l'aide de leur table de multiplication, au chapitre 4, il est d'autant plus important de les présenter ici.

Définition 2.9 *Un* carré latin *est une matrice $n \times n$ dont les cases sont remplies par les éléments d'un ensemble de taille n de telle façon que l'on ne retrouve jamais deux fois le même élément sur une ligne ou sur une colonne.*

Définition 2.10 *Un* carré latin réduit *est un carré latin qui présente sur sa première ligne et sa première colonne les éléments du support en ordre lexicographique.*

Exemple

Voici deux carrés latins ayant comme support $\{0,1,2,3,4\}$. Le carré latin C' est réduit.

$$C = \begin{array}{|c|c|c|c|c|}\hline 0&1&3&4&2\\\hline 1&0&4&2&3\\\hline 3&4&2&1&0\\\hline 4&2&0&3&1\\\hline 2&3&1&0&4\\\hline\end{array} \qquad C' = \begin{array}{|c|c|c|c|c|}\hline 0&1&2&3&4\\\hline 1&0&3&4&2\\\hline 2&3&4&1&0\\\hline 3&4&0&2&1\\\hline 4&2&1&0&3\\\hline\end{array}$$

On remarque que la table de multiplication d'un quasigroupe forme un carré latin et que celle d'une boucle forme un carré latin réduit. En effet, les lois d'annulation à gauche et à droite garantissent l'unicité des éléments sur chaque ligne et chaque colonne de la table de multiplication d'un quasigroupe. De plus, la présence d'un élément neutre chez les boucles donne un élément tel que $ae = ea = a$, donc un carré latin réduit.

Définition 2.11 *Un* rectangle latin *est une matrice rectangulaire de taille $r \times n$, $n \geq r$, dont les cases sont remplies par les éléments d'un ensemble de taille n, de telle façon que l'on retrouve chaque élément une seule fois sur chaque ligne et que l'on ne retrouve pas deux fois le même élément sur une colonne.*

Exemple

Voici un rectangle latin ayant comme support $\{0, 1, 2, 3, 4\}$ *:*

$$R = \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 2 \\ \hline 2 & 3 & 4 & 1 & 0 \\ \hline \end{array}$$

Tous les carrés et rectangles latins que nous utilisons sont réduits, nous ne le mentionnons donc plus. Le prochain théorème est fondamental pour notre étude. En effet, il nous permet de simplifier le modèle que nous utilisons pour comprendre l'algorithme de base du chapitre 4. Ce théorème est tiré de [15] qui est un cas spécial du "Hall Marriage Theorem" [10].

Théorème 2.1 *On peut toujours ajouter une ligne à un rectangle latin $r \times n$, $n > r$, pour obtenir un rectangle $n \times (r + 1)$.*

Pour établir ce théorème, nous utilisons une représentation par les graphes. Nous fournissons d'abord les outils nécessaires à la section 2.3.1 et ensuite nous présentons la preuve qui se divise en plusieurs parties à la section 2.3.2. Celle-ci est largement inspirée de celle présentée dans [15] aux pages 108 et suivantes, tout en corrigeant une légère erreur.

2.3.1 Les graphes bipartis

Définition 2.12 *Un* graphe non orienté $G(N, A)$, *est formé d'un ensemble de noeuds* N *et d'un ensemble d'arêtes* $A \subseteq N \times N$, *tel que* $(a, b) \in A$ *si et seulement si* $(b, a) \in A$.

Définition 2.13 *Deux noeuds* a *et* b *sont* adjacents *dans* $G(N, A)$ *s'il existe un couple* (a, b) *dans* A.

Définition 2.14 *Un* graphe biparti *est un graphe* $G(N, A)$ *où il est possible de partitionner* N *en deux classes disjointes* N_1 *et* N_2 *de sorte que deux éléments de la même classe ne sont jamais adjacents.*

Définition 2.15 *Un* 1-facteur *d'un graphe* $G(N, A)$ *est un sous-graphe* $G_1(N, A')$ *tel que* A' *est un sous-ensemble de* A *de sorte que chaque noeud n'a qu'une seule arête incidente.*

Définition 2.16 *Un graphe* $G(N, A)$ *est dit* 1-factorisable *si l'ensemble* A *peut être partitionné en sous-ensembles disjoints* $A_1, ..., A_k$, *de telle façon que* $G_i(N, A_i)$ *est un 1-facteur de* $G(N, A)$ *pour tout* $i \in \{1, ..., k\}$.

Définition 2.17 *Dans un graphe biparti* $G(N_1 \cup N_2, A)$, *prenons* $T \subseteq N_1$. *On définit* $img(T)$, *comme* $\{b \in N_2 | (a, b) \in A \text{ et } a \in T\}$.

Exemple

Prenons le graphe $G(N, A)$ *de la figure 2, avec* $N = \{\{a_i\} \cup \{b_i\}, 1 \leq i \leq 5\}$.

Le graphe de la figure 2 est clairement biparti puisque l'on peut diviser les noeuds en deux sous-ensembles disjoints de noeuds non adjacents $\{a_i\}$ *et* $\{b_i\}, 1 \leq i \leq 5$. *Si on prend le graphe n'ayant que les arêtes en lignes pleines, on construit un 1-facteur de ce graphe. On peut faire la même chose avec les lignes pointillées. Puisque l'on peut partitionner ce graphe en deux 1-facteurs disjoints, il est 1-factorisable.*

Si on prend $T = \{a_1, a_3\}$, *alors* $img(T) = \{b_1, b_2, b_3, b_5\}$.

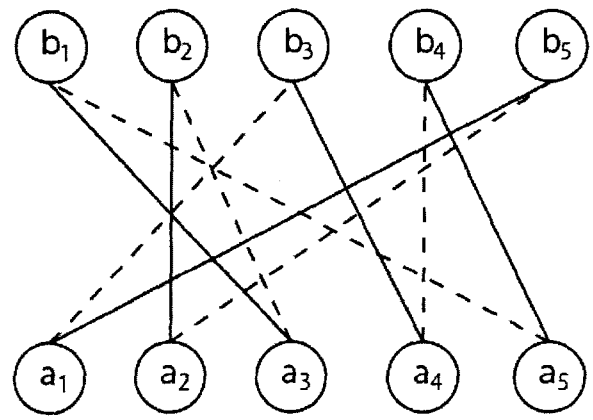

FIG. 2: Graphe biparti G

2.3.2 On peut compléter un rectangle latin

Pour la preuve du théorème 2.1, nous utilisons un graphe biparti bien précis. Commençons donc par en décrire la construction en détail.

Définition 2.18 *Soit un graphe biparti $G_R(N, A)$ construit à partir d'un rectangle latin R de la façon suivante :*

1. *N se partitionne en N_1 et N_2 où N_1 représente les colonnes du rectangle latin et N_2 représente les éléments du rectangle latin R.*
2. *L'arête $(a, b) \in A$ si et seulement si la colonne a ne contient pas l'élément b.*

Le graphe de la figure 2 a été construit par cette méthode sur le rectangle latin du tableau 1.

Lemme 2.1 *Chaque 1-facteur $G'_R(N, A')$ de $G_R(N, A)$ permet d'ajouter une ligne au rectangle latin de R.*

Preuve Il suffit de prendre chaque élément (a, b) de A' et de placer l'élément b dans la colonne a. Comme on a un 1-facteur, on est certain de trouver une seule fois chaque colonne et une seule fois chaque élément. □

1	3	4	2	5
4	1	3	5	2
2	4	5	1	3

TAB. 1: Rectangle latin

Lemme 2.2 *Soit R un rectangle latin $r \times n$ et soit S_i l'ensemble des éléments manquants sur la i^{me} colonne R, $1 \leq i \leq n$.*

Si $T \subseteq \{1, 2, ..., n\}$ et $S = \bigcup_{i \in T} S_i$, alors $|S| \geq |T|$.

Preuve Il faut prouver que dans k colonnes quelconques d'un rectangle latin, $k \leq n$, il manque au moins k symboles. Prenons T un sous-ensemble quelconque de $\{1, 2, ..., n\}$ et R un rectangle latin $r \times n$. Dans les $|T|$ colonnes choisies, il manque $|T|(n - r)$ éléments. Il faut prouver que dans ces $|T|(n-r)$ cases, il manque au moins $|T|$ symboles différents. Supposons le contraire et qu'il manque $l < |T|$ symboles distincts. Alors, on ne pourra ajouter que $l(n - r)$ éléments dans les $|T|(n - r)$ cases vides puique l'on ne peut ajouter le même élément deux fois sur la même ligne. Puisque $l < k$, il restera des cases libres et donc l'hypothèse $l < r$ est fausse. □

Lemme 2.3 *Le graphe biparti $G_R(N, A)$ est 1-factorisable.*

Preuve Soit $G(N_1 \cup N_2, A)$ un graphe biparti quelconque.

Prenons $l \in \{1, 2, ..., n\}$ et $S \subseteq N_1$ de façon à ce que $|S| = l$, $n = |N_{1|}|$.

Définissons :

1. $P_1(S)$ est vrai $\Leftrightarrow \forall T \subseteq S, |T| \leq |img(T)|$.
2. $P_2(S)$ est vrai $\Leftrightarrow$ La restriction de G aux noeuds $S \cup img(S)$ possède un 1-facteur.

Il manque au moins k éléments à k colonnes quelconques d'un rectangle latin selon le théorème 2.2. Ce qui signifie que $P_1(S)$ est vrai pour notre graphe G_R. Nous voulons prouver que $P_1(S) \Rightarrow P_2(S)$ pour tout graphe biparti. Procédons par induction mathématique sur la taille l de S.

Base : Prenons $l = 1$.

Si on prend un seul noeud u d'un graphe biparti et que $img(u) \geq 1$, alors on peut trouver un 1-facteur en choisissant une des arêtes incidentes à u.

Hypothèse d'induction : Supposons que $P_1(S) \Rightarrow P_2(S)$, pour $|S| \leq l$ est vrai.

Pas d'induction : Prenons $|S| = l + 1$ tel que $P_1(S)$ est vrai.

Nous avons deux cas à traiter :

1. $\forall T \subset S, |T| < |img(T)|$:

   Prenons $u \in S$ et $w \in img(u)$ et posons $S' = S - \{u\}$.

   Alors $\forall T \subseteq S', |T| \leq |img(T) - \{w\}|$.

   Donc, $P_1(S')$ est vrai et $P_2(S')$ est vrai par l'hypothèse d'induction.

   Il suffit d'ajouter (u, w) au 1-facteur que nous savons exister puisque $P_2(S')$ est vrai.

   On a ainsi un 1-facteur de G restreint aux noeuds $S \cup img(S)$ et $P_2(S)$ est vrai.

2. $\exists T \subset S, |T| = |img(T)|$:

   On sait que $P_1(T)$ est vrai et donc par l'hypothèse d'induction, $P_2(T)$ est vrai.

   Soit G' le graphe auquel on enlève les noeuds $T \cup img(T)$ et toutes les arêtes incidentes à l'un de ces noeuds.

Prenons $S' = S - T$. Nous allons maintenant définir P_1' et P_2' respectivement les propriétés P_1 et P_2 restreintes au graphe G'.

Nous voulons prouver que $P_2'(S')$ est vrai.

Si $P_1'(S')$ est faux, alors il existe $X \subseteq S'$ tel que $|X| > |img(X) - img(T)|$. Cela implique que $|X \cup T| > |img(X \cup T)|$ contredisant l'hypothèse initiale qui dit que $P_1(S)$ est vrai.

Nous savons que $P_1'(S')$ est vrai, car $|S| \leq |img(S)|$ et on enlève le même nombre d'éléments dans S et dans $img(S)$. Donc, par hypothèse d'induction, $P_2'(S')$ est vrai et $P_2(T \cup S')$ est vrai.

□

Nous allons utiliser ce théorème à la section 4.2 pour simplifier la représentation de l'algorithme de génération.

2.4 Les langages

Il existe un lien étroit entre les langages formels et les groupoïdes (voir [3]). Même si ce lien ne semble pas évident au premier abord, une théorie les relie en plusieurs points. Puisque c'est ce lien qui motive une grande partie de notre recherche, il nous semble important de montrer les principaux résultats de ce domaine. Pour ce faire, nous présentons ici quelques définitions. Ensuite, nous définissons deux catégories de langages : les langages réguliers et les langages hors-contextes. La section suivante fait les liens entre ces langages et différents automates.

2.4.1 Définitions

Commençons par les définitions de base qui permettent de définir les langages réguliers et hors-contextes.

Définition 2.19 *Un* alphabet *est un ensemble fini non vide d'objets appelés symboles.*

Définition 2.20 *Une séquence finie de symboles sur un alphabet est appelée un* mot.

Définition 2.21 *Un* langage *est un ensemble fini ou infini de mots sur un alphabet.*

Définition 2.22 *Le* mot vide *noté ε, est le mot de longueur* 0.

Par exemple, on peut choisir $\Sigma = \{0, 1\}$. Sur cet alphabet, on peut définir le mot 110011. On pourrait également définir L_1 comme étant le langage de tous les mots de taille 4 ou encore L_2 les mots ne comprenant que des 1.

Certaines opérations sur les langages permettent de les définir plus précisément et plus simplement. Plus important encore, ces opérations permettent de définir des familles de langages.

Les opérations d'union, d'intersection, de différence symétrique et de complémentation sont naturelles sur les langages puisque ceux-ci sont des ensembles. Les deux opérations suivantes sont définies formellement pour éviter toute ambiguïté.

Définition 2.23 *Soit deux langages L_1 et L_2. La* concaténation *de ces deux langages, $L_1 \circ L_2$, sera le langage $L = \{xy$ tel que $x \in L_1$ et $y \in L_2\}$.*

Notons que si $L_1 = \emptyset$ et L_2 est un langage quelconque, $L_1 \circ L_2 = \emptyset$. En effet, en se rapportant à la définition précédente, on voit qu'il n'y a aucun couple xy, puisque L_1 est vide.

L'étoile de Kleene ou opération étoile est extrêmement importante pour la définition de langages et de familles de langages.

Définition 2.24 *Soit L un ensemble, alors on définira récursivement L^* de la façon suivante :*

1. $L^0 = \{\varepsilon\}$
2. $L^k = L \circ L^{k-1}$, $k > 0$
3. $L^* = \bigcup_{k \geq 0} L^k$

Par exemple, si le langage $L = \{00\}$ alors $L^* = \{\varepsilon, 00, 0000, 000000, ...\}$.

2.4.2 Les langages réguliers

Les langages réguliers sont ceux construits à partir uniquement des trois opérations de base soit l'union, la concaténation et l'opération étoile.

Définition 2.25 *Soit Σ un alphabet. La classe des* langages réguliers *sur Σ est définie inductivement de la façon suivante :*

1. *$\emptyset$ est un langage régulier.*
2. *Pour tout $\sigma \in \Sigma$, $\{\sigma\}$ est un langage régulier.*
3. *Si L_1 et L_2 sont des langages réguliers, alors $L_1 \cup L_2$ est régulier.*
4. *Si L_1 et L_2 sont des langages réguliers, alors $L_1 \circ L_2$ est régulier.*
5. *Si L est un langage régulier, L^* l'est aussi.*

Exemple

Sur l'aphabet $\{0, 1\}$, les langages suivants sont réguliers :

- *$\emptyset$ est un langage régulier, par la règle 1.*
- *$\{\varepsilon, 0^2, 0^4, 0^6, ...\}$ est un langage régulier. En effet, $\{0\}$ est un langage régulier par la règle 2. Donc $\{00\}$ est aussi un langage régulier par la règle 4. Par la règle 5, $\{00\}^*$ est aussi un langage régulier.*
- *$\{\varepsilon\}$ est un langage régulier puisque $\emptyset^* = \{\varepsilon\}$.*

2.4.3 Les langages hors-contextes

Définition 2.26 *Une* grammaire hors-contexte *est un 4-uple* $G = (N, \Sigma, P, S)$, *où*

1. N *est un alphabet appelé l'ensemble des non-terminaux.*
2. Σ *est un alphabet appelé l'ensemble des terminaux, avec* $N \cap \Sigma = \emptyset$.
3. P *est un ensemble fini de règles de substitutions de la forme* $A \longrightarrow w$, *où* $A \in N$ *et* $w \in (N \cup \Sigma)^*$.
4. S *est le symbole de départ.*

Si on a une grammaire G, on notera $L(G)$ le langage généré par cette grammaire. Prenons comme exemple le langage $L(G_{(\,)})$ sur l'alphabet $\{(,)\}$ qui comprend tous les mots bien parenthétisés. Définissons $G_{(\,)} = (N, \Sigma, P, S)$ la grammaire associée à $L(G_{(\,)})$. Nous avons

$N = \{D\}$

$\Sigma = \{(,), \varepsilon\}$

$P = \{D \longrightarrow \varepsilon, D \longrightarrow DD, D \longrightarrow (D)\}$

$S = D$.

En appliquant ces règles dans n'importe quel ordre, nous constatons que seuls des mots bien parenthétisés sont générés.

Définition 2.27 *Un langage* L *est hors-contexte s'il existe* G, *une grammaire hors-contexte, telle que* $L = L(G)$.

2.5 Les automates

Au cours de cette section nous présentons quelques modèles de calcul mathématique. Plus particulièrement nous présentons l'automate fini déterministe (AFD) de même que l'automate à pile sous ses formes déterministe (APD) et non-déterministe (APND).

Les langages reconnu par ces automates sont exposés à la suite de la définition formelle de chaque automate. Il est important de comprendre les liens entre les automates et les langages qu'ils reconnaissent pour saisir la motivation de cette recherche qui clôt le chapitre.

2.5.1 Les automates finis (déterministes)

Définition 2.28 *Un* automate fini déterministe *(*AFD*) est un 5-uple* $(Q, \Sigma, \delta, q_0, F)$ *où :*

1. Q *est un ensemble fini, non vide d'éléments appelés états.*
2. Σ *est un alphabet d'entrée.*
3. δ *est la fonction de transition* $\delta : Q \times \Sigma \mapsto Q$.
4. $q_0 \in Q$ *est l'état initial.*
5. $F \subseteq Q$ *est l'ensemble des états acceptants.*

Un AFD répond à la question suivante : «Est-ce que l'entrée est acceptée ou refusée?» L'automate accepte si, après la lecture complète de son entrée, l'état où il se retrouve appartient à l'ensemble F des états acceptants. Dans le cas contraire, on dit que l'automate rejette l'entrée.

Un automate fini fonctionne de la façon suivante :

1. Le AFD commence dans l'état q_o.
2. Une lettre du mot d'entrée est lue.
3. Si l'entrée n'est pas entièrement lue,

   La fonction δ est appliquée selon l'état présent et la lettre qui a été lue en 2 pour arriver dans un nouvel état.

   Retour à l'étape 2.

4. Sinon,

   Si l'état présent appartient à F, l'automate accepte l'entrée.

   Sinon l'automate rejette l'entrée.

Exemple

Définissons un AFD qui reconnaît, sur l'alphabet $\{0,1\}$*, les mots qui ont un nombre pair de* 1. *Soit* $M = \{\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_0\}\}$. *La fonction de transition est définie par la table suivante :*

# transition	*État courant*	*Symbole d'entrée*	*État suivant*
1	q_0	0	q_0
2	q_0	1	q_1
3	q_1	0	q_1
4	q_1	1	q_0

On peut représenter un AFD par un graphe dirigé. On représente les états par des noeuds, les transitions par des arcs. L'état initial est identifié par une flèche et les états finaux par des doubles cercles.

Exemple

La figure 3 représente le AFD acceptant tous les mots dont le nombre de 1 *est pair.*

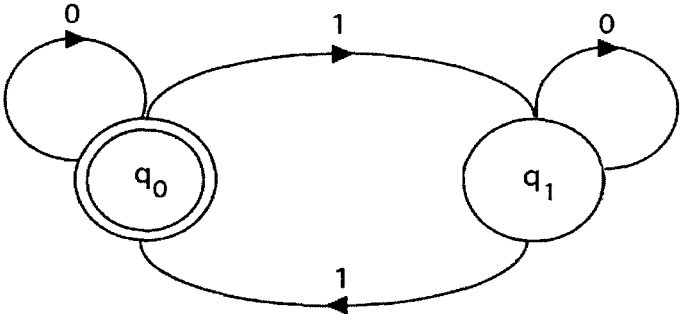

FIG. 3: DFA reconnaissant la parité des 1 dans une chaîne sur un alphabet binaire

Définition 2.29 *L'ensemble des mots reconnus par une machine* $M = (Q, \Sigma, \delta, q_0, F)$ *est appelé le langage accepté par* M *et est noté* $L(M)$.

Théorème 2.2 *(Kleene 1956 [12])L'ensemble des langages reconnus par les automates finis déterministes est exactement l'ensemble des langages réguliers.*

La preuve de ce théorème fondamental dépasse un peu le cadre de notre exposé. Elle est disponible dans tout livre d'introduction à l'informatique théorique. Pour plus de détails nous renvoyons le lecteur à [8].

2.5.2 Les automates à pile (déterministes)

Un AFD ne dispose par définition que d'une quantité finie de mémoire. On peut cependant, à l'aide de rubans de travail qui sont, eux, de taille infinie, augmenter la capacité de calcul d'un automate. C'est ce que fait l'automate à pile non déterministe ou APND. Comme cet automate est une extension de l'automate à pile déterministe (APD), nous allons commencer par définir celui-ci.

Un APD possède, en plus des mêmes caractéristiques qu'un AFD, un ruban de travail limité qui fonctionne comme une pile. Cet ajout augmente sa capacité de calcul.

Définition 2.30 *Un* APD *est un 6-uple* $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, *où*

1. Q *est un ensemble fini non vide d'états.*
2. Σ *est l'alphabet d'entrée.*
3. Γ *est l'alphabet de la pile.*
4. δ *est la fonction de transition* $\delta : Q \times \Sigma' \times \Gamma' \rightarrow Q \times \Gamma'$ *où*

$$\Sigma' = \Sigma \cup \{\varepsilon\}$$

$$\Gamma' = \Gamma \cup \{\varepsilon\}$$

5. q_0 *est l'état initial,* $q_0 \in Q$.
6. F *est l'ensemble des états acceptants,* $F \subseteq Q$.

Décrivons l'interprétation à faire de la fonction δ. Soit $(q_1, a, c) \rightarrow (q_2, d)$, une transition appartenant à δ. Alors, il faut interpréter chaque élément de cette façon :

q_1 est l'état courant,

a est le symbole d'entrée,

c est le symbole à enlever de la pile

q_2 est l'état suivant,

d est le symbole à écrire sur la pile.

Un APD accepte une entrée si, lorsque le mot d'entrée à été lu entièrement, le APD est dans un état acceptant et que la pile est vide.

2.5.3 Les automates à pile (non déterministes)

Un automate à pile non déterministe (APND) est défini comme un APD à l'exception de la fonction de transition qui devient une relation. En effet, à partir d'un état donné, il peut exister plusieurs transitions possibles. Sur une même entrée, plusieurs calculs peuvent donc être effectués par un APND. L'acceptation est également un peu modifiée par rapport à un APD. On dit qu'un APND accepte une entrée si l'un de ses calculs accepte l'entrée. Une fois encore, on peut effectuer un lien direct entre les langages formels et les automates.

Théorème 2.3 *Les APND reconnaissent exactement les langages hors-contextes.*

Une fois de plus la preuve de ce théorème dépasse un peu le cadre de notre exposé. Une preuve qui construit le APND à partir d'un arbre construit sur une grammaire hors-contexte est disponible dans [8] ou dans la plupart des livres d'introduction à l'informatique théorique.

2.6 Reconnaissance de langages par des groupoïdes

On peut utiliser les groupoïdes comme des machines à reconnaître des langages. Selon les propriétés de l'opération binaire du groupoïde, on reconnaît certaines familles de langages. Dans la présente section nous nous penchons sur les langages reconnus par les monoïdes, les groupoïdes et enfin les boucles.

2.6.1 Reconnaissance de langages par des monoïdes

Les monoïdes, groupoïdes associatifs possédant un élément neutre, sont des objets mathématiques définis selon des propriétés algébriques. Cependant, on peut voir un monoïde comme une machine à reconnaître des langages.

Prenons d'abord un exemple qui nous permet de bien comprendre comment un monoïde peut être utilisé comme une machine.

Exemple

Prenons le monoïde M suivant et le sous-ensemble acceptant $\{0\}$.

$$(M,\cdot) = \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$$

Prenons un mot $\omega = 100010001101010$. On peut évaluer l'expression équivalente $1 \cdot 0 \cdot 0 \cdot 0 \cdot 1 \cdot 0 \cdot 0 \cdot 0 \cdot 1 \cdot 1 \cdot 0 \cdot 1 \cdot 0 \cdot 1 \cdot 0$ avec le monoïde M et on arrive à 0. Notons que M est un monoïde et donc associatif, toutes les parenthétisations arrivent donc au même résultat. Si on voit les lignes de la table de multiplication comme des états et les colonnes comme des transitions, on peut remarquer que ce monoïde reconnaît exactement le langage de la parité des 1, soit le même que l'automate de la figure 3. Lorsqu'il n'y a pas de confusion possible quant à l'opération appliquée, on utilise ab pour exprimer $a \cdot b$ et l'opération est la concaténation comme cela est précisé à la section 2.1.

Pour bien définir comment utiliser les monoïdes finis pour reconnaître des langages, certaines définitions doivent d'abord être données.

Définition 2.31 *Le* monoïde libre *dénoté A^* est l'ensemble des mots sur l'alphabet A, l'opération est la concaténation.*

Notons que, selon la définition 2.25, puisque A est fini, A^* est un langage régulier.

Définition 2.32 *Soit A, un alphabet et M, un monoïde. Un* homomorphisme $h : A^* \to M$ *est une fonction telle que $\forall u, v \in A^*, h(uv) = h(u)h(v)$. (Notons que $h(\varepsilon) = 0$.)*

On peut donc définir des langages à l'aide de monoïdes finis de la façon suivante :

Définition 2.33 *Soit A, un alphabet, le langage $L \subseteq A^*$ est reconnu par un monoïde fini S s'il existe un sous-ensemble $F \subseteq S$ et un homomorphisme $h : A^* \rightarrow S$ tel qu'un mot $\omega \in A^*$ appartient à L si et seulement si $h(\omega) \in F$.*

L'homomorphisme sert à prendre un mot d'un alphabet quelconque et à le traduire en un mot appartenant au monoïde libre. L'évaluation d'un mot à l'aide d'un monoïde consiste à évaluer l'expression jusqu'à n'avoir plus qu'un seul élément.

Théorème 2.4 *Les langages reconnus par les monoïdes finis sont exactement les langages réguliers.*

La preuve de ce théorème est assez simple si on connaît le théorème de Kleene [12]. On utilise les symboles de l'alphabet du AFD comme des fonctions $Q \rightarrow Q$ et la composition de fonctions comme opération. Puisque la composition de fonctions est associative, on a un semigroupe fini. On ferme l'opération et il est facile par la suite d'introduire un élément neutre pour obtenir un monoïde. Pour construire l'AFD à partir du monoïde, on utilise les éléments du monoïde comme des états et l'alphabet est le même que celui du monoïde. Chaque transition (q, a, q') est telle que $q \cdot a = q'$ dans le monoïde. Comme le nombre d'états et l'alphabet sont finis, c'est bien un AFD.

2.6.2 Reconnaissance de langages par des groupoïdes

On peut utiliser les groupoïdes pour reconnaître des langages comme on le fait avec les monoïdes. Par contre, comme il y a plusieurs façons d'évaluer une expression selon la parenthétisation dans un groupoïde, il y a quelques modifications à faire.

Définition 2.34 *Soit un alphabet A, le langage $L \subseteq A^* - \{\varepsilon\}$ est reconnu par un groupoïde fini G s'il existe un sous-ensemble $F \subseteq G$ et un morphisme alphabétique $h : A^* \rightarrow G^*$ de telle façon qu'un mot $\omega \in A^* - \{\varepsilon\}$ appartient à L si et seulement si $h(\omega)$ peut s'évaluer à un élément de F.*

Exemple

Le groupoïde reconnaissant le langage L sur l'alphabet $A = \{(,)\}$ des mots bien parenthétisés.

Soit le groupoïde dont la table de multiplication est :

$\cdot$	0	1	2	3
0	0	1	2	3
1	1	3	0	3
2	2	3	3	3
3	3	3	3	3

Prenons le morphisme $\Phi : A \rightarrow G$ suivant :

$$\Phi(() = 1$$
$$\Phi()) = 2$$

On peut étendre de façon naturelle ce morphisme à $h : A^* \rightarrow G^*$. Prenons $F = \{0\}$ comme ensemble acceptant. En évaluant $h(\omega)$ dans G, on reconnaît le langage L par les mots qui s'évaluent à 0.

Prenons le mot $\omega =$ (() ()). Alors $h(\omega) = 112122$. Devinons la bonne parenthétisation qui évalue $h(\omega)$ à 0.

1	112122
2	1(12)(12)2
3	1002
4	1(00)2
5	12
6	0

Donc il existe une façon d'évaluer $h(\omega)$ à 0. Comme $0 \in F$, l'ensemble acceptant, le mot était bien parenthétisé.

Notons que G doit posséder un élément neutre (0) pour reconnaître le mot vide. On définit alors $h(\varepsilon) = 0$. Par contre, si G ne possède pas d'élément neutre, le mot vide ne peut alors être reconnu. Il arrive ainsi que le domaine de h devienne $A^+ = A^* - \varepsilon$.

Théorème 2.5 *Les groupoïdes reconnaissent exactement les langages hors-contextes.*

Une fois de plus c'est le résultat et non la preuve qui nous intéresse ici. La preuve originale a été publiée et est disponible dans [3].

2.6.3 Reconnaissance de langages par des boucles

On peut également utiliser les boucles pour reconnaître des langages sur A^*. On le fait exactement de la même façon qu'avec les groupoïdes, l'opération n'étant pas associative. On définit donc, pour une boucle B, un sous-ensemble $F \subseteq B$ des éléments acceptants et un morphisme du monoïde libre vers une séquence d'éléments de la boucle.

L'utilisation des boucles pour reconnaître des langages est une idée assez récente. On connaît les familles de langages reconnues par celles-ci. Cependant, aucune théorie supplémentaire ne permet de partitionner ces familles selon les propriétés algébriques des boucles les reconnaissant. Le théorème 2.6 se rapporte aux langages réguliers ouverts qui sont reconnus par les boucles, mais quelques définitions tirées de [27] s'imposent d'abord.

Définition 2.35 *Soit un alphabet A, définissons la fonction $g : A^* \times A^* \to \mathbb{N}$ telle que $g(x, y)$ est l'ordre du plus petit groupe G pour lequel il existe un homomorphisme $\varphi : A^* \to G$, tel que $\varphi(x) \neq \varphi(y)$.*

Définition 2.36 *Pour $x, y \in A^*$, définissons $d : A^* \times A^* \to \mathbb{R}$ la* distance *entre x et y, $d(x, y) = e^{-g(x,y)}$.*

Le lecteur peut vérifier que $d(x, y)$ est bien une distance au sens mathématique en se référant à [17]. La distance d définit donc un espace métrique à partir duquel on peut définir des langages réguliers ouverts. Le théorème suivant (voir [27]) permet de définir les langages réguliers ouverts d'une toute autre façon.

Théorème 2.6 *Les* langages réguliers ouverts *sont des langages de la forme $a_1 L_1 \circ a_2 \circ L_2 \circ ... \circ L_k \circ a_{k+1}$, où les $L_i, 1 \leq i \leq k$, sont des langages reconnus par des groupes; les a_i, des lettres.*

Théorème 2.7 *[1] Les boucles reconnaissent exactement les langages réguliers ouverts.*

Ce dernier résultat permet de comprendre que les boucles et les langages sont des concepts très liés. Par exemple, il a été prouvé que les langages réguliers ouverts sont des langages de la forme $L_1 \circ L_2 \circ ... \circ L_k$, où les $L_i, 1 < i < k$, sont des langages reconnus par des groupes et sont appelés *langages à groupes.* En passant par les boucles, il a été prouvé dans [1] que cette famille de langages est la même que celle définie par le théorème 2.6.

2.6.4 Les langages reconnus par des boucles isotopiques et isomorphes

Nous nous intéresserons aux boucles isomorphes dans cette recherche alors que nombre de chercheurs dans le domaine des boucles s'intéressent aux classes d'isotopie. La raison qui nous tourne vers les isomorphismes est tirée directement de l'utilisation que nous faisons des boucles, la reconnaissance de langages.

Théorème 2.8 *Si L est un langage reconnu par une boucle B, alors L est reconnu par toute boucle B' isomorphe à B.*

Preuve La preuve est assez simple en se reportant directement aux définitions. Soit une boucle B, un alphabet A et un langage $L \subseteq A^*$ reconnu par B avec le morphisme $h : A^* \to B^*$ et F un ensemble acceptant. Alors prenons la boucle B' isomorphe à B par l'isomorphisme $\alpha : B \to B$. En prenant le morphisme $h' : A^* \to B'^*$ de telle façon que $h'(a) = \alpha(h(a))$ et l'ensemble $F' = \{a \in B', \alpha^{-1}(a) \in F\}$, B' reconnaît le même langage puisque α est un isomorphisme. □

Par contre, si deux boucles sont isotopiques, il est possible qu'elles ne reconnaissent pas les mêmes langages. Nous procédons ici par un contre-exemple qui présente deux boucles isotopiques. L'une reconnaît un langage et l'autre non.

Exemple

Prenons d'abord la boucle suivante :

$(B,+) =$

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	0	3	4	2
2	2	3	4	1	0
3	3	4	0	2	1
4	4	2	1	0	3

Nous allons considérer l'alphabet $A = \{a, b\}$ *et le langage* $L = A^*bA^*$. *Ce langage est l'ensemble des mots sur* A *qui ont au moins un* b. *Ce langage est reconnu par la boucle* B *de la façon suivante :*

1. *Prenons* $h : A^* \to B^*$ *tel que* $h(a) = 0$ *et* $h(b) = 2$.
2. *L'ensemble acceptant* F *sera* $B - \{0\}$.

Alors, si un mot $\omega = a^*$ *ne comprend pas de* b, *alors* $h(\omega) = h(a)h(a)...h(a) = 0$, *et* $eval(\omega) \notin F$ *donc* ω *est rejeté. Par contre, si nous prenons le mot* ω' *qui comprend*

des b et considérons que l'on peut toujours évaluer une suite de a de ω' à 0. Alors :

Si ω' n'a qu'un seul b, alors $h(\omega) = h(b) = 2$ et le mot est accepté.

Si ω' a deux b, $h(\omega') = h(b) + h(b) = 2 + 2 = 4$ et le mot est accepté.

Si ω' en a trois, $eval(h(\omega')) = \{0, 1\}$ et le mot est accepté.

Si ω' a plus de trois b, $|eval(h(\omega))| > 2$, puisque $eval(h(\omega'))$ a plus de deux éléments et que B respecte la loi d'annulation.

Considérons maintenant la boucle suivante :

$$(B,\cdot) = \begin{array}{c|ccccc} \cdot & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 2 & 3 & 4 \\ 1 & 1 & 0 & 3 & 4 & 2 \\ 2 & 2 & 4 & 0 & 1 & 3 \\ 3 & 3 & 2 & 4 & 0 & 1 \\ 4 & 4 & 3 & 1 & 2 & 0 \end{array}$$

Nous allons prouver que cette boucle ne peut reconnaître le langage $L = A^*bA^*$, $A = \{a, b\}$. Pour ce faire, nous allons faire une preuve par l'absurde. Considérons la composition de F l'ensemble acceptant. Alors 0 ne peut être dans l'ensemble acceptant parce que $h(\varepsilon) = 0$ et $\varepsilon \notin L$. De plus, si 0 est dans l'ensemble acceptant, alors $h(aa) = h(a)h(a) = 0$ et on accepte un mot qui n'appartient pas à L. Donc $(B, \cdot)$ ne reconnaît pas le langage L.

On a vu à l'exemple de la page 15 que les deux boucles $(B, +)$ et $(B, \cdot)$ sont isotopiques. Or, nous venons de prouver qu'il y a au moins un langage qui n'est pas reconnu par les deux. Les classes d'isotopies ne sont donc pas suffisantes pour l'utilisation que nous faisons des boucles, soit la reconnaissance de langages.

2.7 Motivation de cette recherche

Nous allons expliquer le cheminement de notre recherche dans cette section en deux parties. La première explique les raisons pour lesquelles nous nous sommes intéressé aux classes d'isomorphisme de boucles, plutôt qu'aux boucles ou aux classes d'isotopie. La seconde motive notre choix de développer un algorithme plutôt que d'utiliser une méthode existante et présente le niveau de complexité du problème.

La classification des groupes a été totalement achevée en 1980 (voir [28]). Tout groupe appartient ainsi à une des classes de groupes définies par le théorème de classification. Il a fallu, pour arriver à ce classement, un effort concerté des mathématiciens semblable à celui des biologistes pour le décodage du génome humain.

En comparaison, il existe chez les semigroupes et les langages réguliers une classification que l'on appelle variété. Le théorème des variétés d'Eilenberg, qui se retrouvere dans [26] ou originalement dans [7], dit qu'il existe une bijection entre les variétés de monoïdes finis et les variétés de langages réguliers.

Il a été possible dans le passé d'étudier les monoïdes, les semigroupes et les groupes en raison de l'associativité. Une large théorie existe sur les monoïdes et sur leurs liens avec les langages formels (voir [25]). Par exemple, on peut classer les langages réguliers selon les propriétés algébriques des monoïdes qui les reconnaissent.

Exemple

Un monoïde M est dit apériodique s'il existe un entier n tel que pour tout élément $a \in M$ nous avons que $a^n = a^{n+1}$. Les langages sans étoile, ceux que l'on peut définir à l'aide de la concaténation et des opérateurs booléens, sont reconnus par les monoïdes apériodiques.

On ne connaît pas chez les groupoïdes non associatifs ce genre de propriétés. On sait que les groupoïdes reconnaissent exactement les langages hors-contextes (voir [3]),

mais on n'a pas d'outils algébriques pour arriver à classifier ces langages. Afin de se donner un point de départ, et parce qu'elles sont plus simples à étudier que les groupoïdes, nous croyons intéressant de mieux connaître les boucles.

Notre approche pour le faire est la même que celle de nos prédécesseurs avec les semigroupes et les groupes : une étude exhaustive. Comme les boucles sont très nombreuses (voir tableau 2), nous étudions des classes de boucles. Les classes d'isotopie permettent ce genre de classement. Malheureusement, comme nous l'avons mentionné à la section 2.2.2, deux boucles isotopiques ne reconnaissent pas nécessairement les mêmes langages. Nous nous sommes donc penché sur une autre classification classique : les classes d'isomorphisme.

Taille	Nombre de boucles	nombre de classes d'isomorphisme	Nombre de groupes
1, 2, 3	3	3	3
4	4	2	5
5	56	6	14
6	9408	109	51
7	16 942 080	23 746	267
8	535 281 401 846	106 228 849	2328
9	377 597 570 964 258 816	9 365 022 303 540	56 092

TAB. 2: Tableau de complexité

Nous n'avons pas trouvé d'algorithme efficace pour générer les classes d'isomorphisme de boucles. La génération d'objets combinatoires non isomorphes est un domaine de recherche conn (voir [19]). Cependant, les algorithmes existants sont génériques et ne traitent que très peu du calcul d'isomorphisme. De plus, bien que le

nombre de boucles et de classes d'isomorphisme de boucles soit connu pour des ordres inférieurs à 10 (voir [20]) nous n'avons pas pu analyser l'algorithme utilisé par les auteurs.

Nous avons donc développé un algorithme. Pour commencer, nous avons étudié en profondeur les propriétés algébriques et combinatoires des boucles de taille 5 à 7 pour arriver à concevoir un algorithme générant toutes les classes d'isomorphisme de boucles de taille n.

Le tableau 2 montre le nombre de boucles, de classes d'isomorphisme de boucles et le nombre de groupes pour des tailles de 1 à 9. Les nombres de boucles sont tirées de [21], les nombres de classes d'isomorphisme d'ordre 1 à 7 étaient plutôt simples à calculer et étaient connus. Elles peuvent être trouvées en effectuant une recherche sur le site [22]. Pour ce qui est des nombres de classes d'isomorphisme d'ordre 8 et 9, ils sont tirés de [20]. Nous avons pu vérifier tous ces résultats de façon indépendante jusqu'à l'ordre 7, de même que le nombre de classes d'isomorphisme d'ordre 8.

CHAPITRE 3

ISOMORPHISME DE BOUCLES

Nous allons, au cours de ce chapitre, fournir tous les outils théoriques qui permettent de comprendre les algorithmes que nous présentons au chapitre 4. Nous allons commencer par définir les permutations qui sont intimement liées aux lignes et aux colonnes des carrés latins. Ensuite, nous définissons les relations d'ordre qui nous permettent de comparer des boucles, des permutations et des cycles. Nous terminons par une étude des représentants de classes d'isomorphisme de boucles et plus particulièrement de leur deuxième ligne.

Plus en détail, nous commençons à la section 3.1 par définir les permutations. Nous voyons notamment comment les écrire sous plusieurs représentations. Nous voyons de plus comment elles sont liées de près à l'étude des boucles par le biais des carrés latins.

Ensuite, à la section 3.2, nous parlons des relations d'ordre. Nous définissons le concept de façon générale sur les ensembles. Par la suite nous posons des relations d'ordre appliquées aux boucles et aux permutations.

Enfin, à la section 3.4, nous étudions en profondeur la deuxième ligne des représentants de classes d'isomorphisme de boucles. Particulièrement, nous nous intéressons aux cycles de ces permutations et aux propriétés des cycles, en relation avec les isomorphismes. Nous arrivons ainsi à caractériser parfaitement les deuxièmes lignes de tout représentant de classes d'isomorphisme de boucles.

En conclusion, nous introduisons le chapitre 4 qui utilise les notions de ce chapitre pour générer les classes d'isomorphisme de boucles pour les ordres inférieurs à 9.

3.1 Permutations

Les permutations sont des objets mathématiques qui ont été largement étudiés. On connaît particulièrement les groupes de permutations. Nous ne nous intéressons ici qu'à la définition et à la notation des permutations (voir [9]). Pour plus de détails, voir [9] sur permutations et les groupes de permutations.

Définition 3.1 *Une* permutation *est une fonction bijective d'un ensemble dans lui-même.*

Sous l'ensemble $S = \{0, 1, 2, 3, 4, 5\}$, $\pi = \{(0,4), (1,2), (2,3), (3,1), (4,5), (5,0)$ est une permutation de S. On voit immédiatement qu'il y a, pour un ensemble de n éléments, $n!$ permutations différentes des éléments de cet ensemble.

Différentes représentations permettent de représenter les permutations. On utilise celle qui sert le mieux le problème. Comme une permutation est une fonction bijective, chacune des représentations est une définition de cette fonction en énumération. La première représentation est celle *point à point.* Elle consiste à définir la permutation dans un tableau où la première ligne représente le domaine et la seconde, l'image. Comme le domaine est toujours l'ensemble des valeurs du support en ordre lexicographique, on écrit souvent seulement l'image. Cette représentation est appelée *point à point abrégée.*

Exemple

On pourrait représenter la permutation π point à point de la façon suivante :

$$\pi = \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 4 & 2 & 3 & 1 & 5 & 0 \\ \hline \end{array}$$

ou en point à point abrégé nous obtenons $\pi = 423150$. Notons que cette dernière représentation ne peut être utilisée si certains éléments du support sont formés de deux ou plusieurs symboles.

Définition 3.2 *Un* cycle *c d'une permutation π est un sous-ensemble minimal du support de π tel que si $a_i \in c$ alors $\pi(a_i) \in c$. Si dans π, $\pi(a_1) = a_2$, $\pi(a_{i-1}) = a_i$, $\pi(a_i) = a_1$, alors on note $c = (a_1, ..., a_i)$.*

La seconde technique de définition d'une permutation est appelée *représentation en cycles.* Elle consiste à représenter une permutation comme un produit de cycles disjoints. Avant de donner un exemple de cette représentation, introduisons les définitions suivantes sur les cycles. Elles seront ensuite largement utilisées.

Définition 3.3 *Dans une permutation* $\pi = \pi_1\pi_2...\pi_k$ *où chaque* π_i $(i > 0)$ *est un cycle, définissons les termes suivants :*

Premier cycle *: cycle qui comprend l'identité, ce cycle est toujours nommé* π_1.

Taille d'un cycle *: nombre d'éléments d'un cycle. Pour un cycle* π_i, *on note* $|\pi_i|$.

Plus petit élément d'un cycle *: élément le plus petit, habituellement le dernier dans la représentation.*

Point fixe *: cycle de taille* 1. *Pour alléger la notation, les points fixes d'une permutation ne sont généralement pas écrits.*

Notons que lorsque nous écrivons une permutation comme un produit de cycles, les cycles sont toujours en ordre croissant en fonction de leur plus petit élément.

Exemple

La permutation π *en cycle prend la forme suivante :* $\pi = (0,4,5)(1,2,3)$. *On dit que* π *a deux cycles de taille 3 et que* $(0,4,5)$ *est le premier cycle. Quant à la permutation* $\pi' = (32)$ *ayant le même support, elle est la même que* $(0)(1)(32)(4)(5)$ *en écrivant les points fixes.*

Il ne faut pas confondre la représentation en cycle et la représentation point à point abrégé. En effet la permutation représentée en cycle $(1,3,2,5,4,0)$ est la même que la permutation représentée en point à point abrégée 135204, et non 132540. Cette ambiguïté se présente lorsqu'il y a un cycle de longueur n, et dans ce cas nous indiquerons clairement la représentation utilisée.

Il est intéressant de voir que chaque ligne et chaque colonne d'un carré latin sont une permutation des éléments de l'ensemble. Cela nous permet de représenter un carré latin C de taille n par n permutations. Nous utiliserons plus loin cette obervation.

Exemple

Le carré latin C suivant peut être défini à l'aide de 5 cycles.

$$C = \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 2 \\ \hline 2 & 4 & 1 & 0 & 3 \\ \hline 3 & 2 & 4 & 1 & 0 \\ \hline 4 & 3 & 0 & 2 & 1 \\ \hline \end{array} \quad = \quad \begin{array}{l} (0)(1)(2)(3)(4) \\ (01)(234) \\ (02143) \\ (03124) \\ (04132) \end{array}$$

3.2 Relations d'ordre et préordre

Pour générer les classes d'isomorphisme de boucles, nous allons comparer des boucles et des permutations de plusieurs façons. La notion mathématique de relation d'ordre permet ce genre d'opération. Nous allons donc, dans la section qui suit, définir les relations d'ordre générales, poser des relations d'ordre pour les boucles et pour les permutations. Finalement, nous utiliserons la relation d'ordre pour définir les représentants de classes d'isomorphisme.

3.2.1 Les relations d'ordre

Une notion naturelle sur les éléments d'un ensemble est celle de relation d'ordre. Bien sûr, pour définir une relation d'ordre sur un ensemble qui soit utile, il faut que celui-ci comprenne au moins deux éléments. Il existe différents types de relations d'ordre, comme nous l'avons mentionné dans [14], qui se catégorisent selon leurs propriétés.

Définition 3.4 *Soit G un ensemble, alors la relation $\leq$ définit un* ordre total *si elle possède les 4 propriétés suivantes :*

$\forall a \in S, a \leq a$ (réflexivité)

$\forall a, b, c \in S, a \leq b \leq c \Rightarrow a \leq c$ (transitivité)

$\forall a, b \in S, a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymétrique)

$\forall a, b \in S, a \leq b \vee b \leq a$ (connectivité)

Si une relation d'ordre ne possède pas la propriété de connectivité, on dit qu'il s'agit d'un *ordre partiel.* Si une relation d'ordre n'est pas antisymétrique, on dit qu'il s'agit d'un *préordre.* On définit également l'*ordre strict* par le symbole $<$. On dit que $a < b$, si et seulement si $a \leq b$ et $a \neq b$.

3.2.2 Ordre des boucles

Un des éléments essentiels pour séparer en classes disjointes les boucles est d'avoir un outil pour les comparer. Nous allons, dans l'algorithme du chapitre 4, générer les boucles selon un ordre préétabli. La relation d'ordre sur les boucles est donc extrêmement importante puisqu'elle détermine l'ordonnancement de génération des boucles.

La relation d'ordre naturelle pour comparer deux boucles est de comparer leur table de multiplication ligne par ligne. Cette relation d'ordre est totale, puisqu'elle est réflexive, transitive, antisymétrique et connectée.

Exemple

Pour comparer les deux boucles suivantes :

$B =$

$+$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$B' =$

$\cdot$	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Il suffit de comparer les lignes une à une :

1. *La première ligne est identique ;*
2. *la seconde ligne de B est plus grande que celle de B'.*

donc, la boule B est plus grande que la boucle B.

Cette relation d'ordre sera appelée à être changée au chapitre 5. En effet, l'algorithme de génération utilisé au chapitre 4 génère en bon ordre les boucles. Pour modifier l'ordre de recherche des isomorphes de boucles, nous modifions la relation d'ordre.

3.2.3 Ordre des permutations

Notre étude nous a poussé à étudier les cycles des permutations. En effet, à la section 3.4 nous utilisons les cycles des permutations que sont les lignes d'un carré latin pour exposer certaines propriétés. Nous avons ainsi à comparer des permutations selon leurs cycles sous un angle bien précis. Cette section définit la fonction qui permet cette comparaison.

Nous comparons les permutations selon deux critères. Premièrement, la longueur du premier cycle (voir définition 3.3) détermine l'ordre de comparaison. Si ce premier

cycle est de même longueur sur les deux permutations, le second critère compare alors les autres cycles.

Pour formaliser le tout, voici la fonction qui construit le mot à comparer à partir d'une permutation.

Définition 3.5 *Soit* Π *l'ensemble des permutations des éléments d'un ensemble* B *et* $\mathbb{N}$ *les nombres naturels. Définissons* $\theta : \Pi \rightarrow \mathbb{N}^n$ *une fonction qui construit pour une permutation* $\pi = \pi_1\pi_2...\pi_k$, $\theta(\pi)$ *de la façon suivante :*

1. $|\pi_1|$, *la longueur du premier cycle, est le premier élément ;*
2. *chaque élément* $i > 0$ *a la valeur* $(n - n_i)$, *où*

   n *est le nombre d'éléments du support,*

   n_i *est le nombre de cycles de taille* i.

Ce vecteur permet une comparaison simple par la suite. En effet, pour comparer deux permutations π_1, π_2, il suffit de comparer en considérant l'ordre naturel des entiers $\theta(\pi_1)$ et $\theta(\pi_2)$.

Exemple

Prenons deux permutations $\pi_1 = (350)(126)(47)$ *et* $\pi_2 = (120)(43)(675)$. *Alors on calcule* $\theta(\pi_1) = 38768888$ *et* $\theta(\pi_2) = 38768888$.

La relation d'ordre sur les vecteurs calculés par θ est totale, puisque l'ordre naturel sur les entiers est total. Regardons une à une les trois propriétés liées aux ordres. Tout d'abord pour la réflexivité, il est clair que $\forall\pi, \theta(\pi) \leq \theta(\pi)$. De même, θ étant une fonction et l'ordre des nombres naturels étant transitif, notre ordre est transitif. Finalement, la comparaison de deux vecteurs associés à une permutation est antisymétrique, par contre, la comparaison de deux permutations n'est pas antisymétrique. En effet, il suffit de prendre $\pi_1 = (350)(126)(47)$ et $\pi_2 = (350)(12)(467)$. On voit que $\theta(\pi_1) \leq \theta(\pi_2)$ et $\theta(\pi_2) \leq \theta(\pi_1)$, mais $\pi_1 \neq \pi_2$. On a donc un préordre.

Ajoutons que lorsque nous comparons deux permutations π et π' sans l'utilisation de θ ($\pi \leq \pi'$), nous faisons référence à la comparaison lexicographique en forme point à point des deux permutations. De plus, lorsqu'il n'y a pas de confusion possible, si π est la $l^{ème}$ ligne d'un carré latin, on écrira $\theta(l)$ en place de $\theta(\pi)$.

3.3 Les classes d'isomorphisme de groupoïdes

On peut partitionner tous les groupoïdes d'un support en classes de telle façon que deux groupoïdes sont dans la même classe si et seulement si ils sont isomorphes. On appelle ces classes, *classes d'isomorphisme.*

Il est possible de générer toute une classe d'isomorphisme en utilisant les $n!$ permutations possibles des éléments de G. Ces $n!$ groupoïdes ne sont pas toujours distincts puisque certains sont le produit d'automorphismes. Il n'est pas trivial de prévoir le nombre de groupoïdes distincts dans chaque classe d'isomorphisme, toutefois ce sujet est traité plus loin.

Pour représenter une classe d'isomorphisme, nous choisissons un de ses membres de façon complètement arbitraire. Dans le cadre de ce travail, une relation d'ordre total sera toujours définie sur les groupoïdes utilisés et il sera ainsi toujours possible de les comparer.

Définition 3.6 *Le* représentant *d'une classe d'isomorphisme est le* plus petit *groupoïde G tel que pour tout $G' \sim G$ on a $G \leq G'$, $\leq$ une relation d'ordre total.*

3.3.1 Les représentants de boucles

Nous allons maintenant étudier plus en profondeur les classes d'isomorphisme et les représentants de boucles. Il est important de comprendre immédiatement une propriété de la fonction α qui calcule l'isomorphisme de deux boucles. Premièrement, si α est un renommage des éléments de la boucle, l'élément neutre ne doit pas être renommé.

Théorème 3.1 *Si $\alpha : (G, \cdot) \to (G, \circ)$ est un isomorphisme et $(G,\cdot)$ une boucle, alors $\alpha(e) = e$.*

Preuve Supposons que e et f soient respectivement les éléments neutres de $(G, \cdot)$ et de $(G, \circ)$ deux boucles isomorphes. Alors, nous avons :

$$\begin{aligned} f \circ \alpha(e) &= \alpha(e) \\ f \circ \alpha(e) &= \alpha(e \cdot e) \\ f \circ \alpha(e) &= \alpha(e) \circ \alpha(e) \\ f &= \alpha(e) \end{aligned}$$

□

On voit donc qu'il n'y a pas $n!$ renommages des éléments d'une boucle de taille n, mais au plus $(n-1)!$.

3.4 Étude de la deuxième ligne

Nous allons maintenant étudier en profondeur la deuxième ligne des représentants de classe d'isomorphisme de boucles. Puisque nous parlons tout au long de cette section des tables de multiplication de boucles, nous parlons de carrés latins et non de boucles. De même, nous parlons de *représentant* pour écourter le vocable *représentant d'une classe d'isomorphisme de carrés latins réduits.*

Définition 3.7 *Une deuxième ligne π est dite* minimale *si pour toutes permutations π' telle que $\theta(\pi) = \theta(\pi')$, on a $\pi \leq \pi'$.*

Nous prouvons qu'il n'y a qu'un très petit nombre de deuxièmes lignes minimales par rapport au nombre de représentants, et qu'il est possible de les générer facilement.

Exemple

Les représentants des carrés latins d'ordre 6 *ont comme deuxième ligne l'une des suivantes :*

$$(10)(32)(54)$$
$$(10)(3452)$$
$$(120)(453)$$
$$(1230)(54)$$
$$(123450)$$

Les trois lemmes qui suivent permettent de caractériser parfaitement les deuxièmes lignes minimales. Ils permettent d'énoncer le théorème 3.2 qui constitue un résultat important de notre étude.

Lemme 3.1 *Dans la colonne* c *de la deuxième ligne d'un représentant, on retrouve* $c+1$ *ou le plus petit élément du cycle auquel appartient cet élément.*

Preuve Nous faisons une preuve par induction sur c.

Base : Pour la première colonne, la preuve est triviale puisque nous avons un carré latin réduit.

Hypothèse d'induction : Supposons le lemme vrai pour tout $c > 1$.

Pas d'induction : Soit p le premier élément du cycle contenant c. Supposons que sur la deuxième ligne d'un représentant, la colonne c contient l'élément $c' \neq c+1$ et $c' \neq p$.

Alors c' doit être plus grand que $c+1$ puisque nous savons que :

1. Tous les éléments plus petits que c sont déjà sur la ligne à l'exception de p par hypothèse d'induction ;

2. Nous avons supposé $c' \neq p$;

3. On ne peut mettre c et on a supposé $c' \neq c+1$.

En appliquant une permutation qui échange c' et $c+1$, on réduit la ligne, ce qui contredit la minimalité de celle-ci.

□

Lemme 3.2 *Soit une permutation $\pi_1\pi_2...\pi_k$ se retrouvant sur la deuxième ligne d'un représentant. Alors, $|\pi_i| \leq |\pi_j|$, $1 < i < j \leq k$.*

Preuve Supposons le contraire et qu'une deuxième ligne $l = \pi_1...\pi_i...\pi_j...\pi_k$, avec $|\pi_i| > |\pi_j|$ se retrouve chez un représentant. On peut supposer sans perdre de généralité que l respecte le lemme 3.1. Posons que $|\pi_i| = (r+1, ..., r)$ et $|\pi_j| = (m+1, ..., m)$. Alors en appliquant la permutation $(m, r)...(m + |\pi_j| - 1, r + |\pi_j| - 1)$, on réduit la deuxième ligne et le carré latin n'est pas représentant ; il y a une contradiction. □

Exemple

Soit le rectangle latin suivant :

$$R = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 2 & 3 \\ \hline 1 & 3 & 0 & 2 \\ \hline \end{array}$$

R ne respecte pas le lemme 3.1, puisque la colonne 1 *ne contient ni* 2 *ni* 0 *le plus petit élément du cycle, donc R n'est pas minimal et ne peut avoir d'enfant représentant. Il suffit d'ailleurs d'utiliser la permutation $\pi = 0132$ et le rectangle obtenu est plus petit.*

Soit cet autre rectangle :

$$R' = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & 2 & 0 & 4 & 5 & 3 & 7 & 6 \\ \hline \end{array}$$

Il n'est pas minimal même si on voit que la deuxième ligne respecte la construction du lemme 3.1. En effet, le deuxième cycle est plus grand que le troisième et le lemme 3.2 n'est pas respecté. La permutation $\pi = (6,2)(7,3)$ *réduit ce rectangle en échangeant les cycles.*

Lemme 3.3 *Il n'y a, pour des cycles donnés, qu'une seule deuxième ligne minimale.*

Preuve Ce lemme découle directement de la définition d'une deuxième ligne minimale et de θ. □

3.4.1 Les deuxièmes lignes minimales

Les deuxièmes lignes minimales sont largement utilisées dans les accélérations de notre algorithme et nous leur consacrons cette section.

Lemme 3.4 *Soit deux lignes minimales* l_1, l_2 *représentées en forme point à point abrégée. Alors* $l_1 \leq l_2$ *si et seulement si* $\theta(l_1) \leq \theta(l_2)$.

Preuve

Premièrement : $\Rightarrow$

Si $l_1 = l_2$, la preuve est facile puisque θ est une fonction. Si $l_1 < l_2$, alors il y a deux cas :

1. Le premier cycle de l_1 est plus petit que celui de l_2. Alors $\theta(l_1) < \theta(l_2)$, puisque le premier nombre généré par θ est la longueur du premier cycle.
2. Le premier cycle de l_1 est de même taille que celui de l_2. Alors, on doit comparer le reste. Soit c la première case qui est différente entre l_1 et l_2. Alors, il faut que la colonne c termine un cycle dans l_1 et non dans l_2. On sait que les cycles sont en ordre de taille sur une deuxième ligne minimale grâce au lemme 3.2. Donc,

le cycle comprenant c dans l_1 est plus petit que celui contenant c dans l_2. Soit i la taille du cycle de c sur l_1. Comme les c premières colonnes de l_1 et l_2 sont identiques, il y a le même nombre de cycles plus petits que i sur l_1 et l_2. Il y a par contre assurément au moins un cycle de taille i de plus sur l_1 que sur l_2. Donc $i^{ème}$ élément de $\theta(l_1)$ est plus petit que celui de $\theta(l_2)$ et $\theta(l_1) \leq \theta(l_2)$.

Deuxièmement : $\Leftarrow$

Si $\theta(l_1) = \theta(l_2)$, alors la preuve est simple puisqu'il y a, pour des cycles donnés, qu'une seule deuxième ligne minimale, donc $l_1 = l_2$. Supposons donc $\theta(l_1) < \theta(l_2)$. Il y a deux cas :

1. Si le premier élément de $\theta(l_1)$ est plus petit que le premier de $\theta(l_2)$, alors 0 se retrouve sur une colonne plus petite sur l_1 que sur l_2 et $l_1 < l_2$.
2. Si le premier élément de $\theta(l_1)$ est le même que celui de $\theta(l_2)$. Soit i le premier élément de $\theta(l_1)$ qui est plus petit sur $\theta(l_2)$. Alors on sait qu'il y a plus de cycles de taille i sur $\theta(l_1)$ que sur $\theta(l_2)$, les autres plus petits étant égaux, donc $l_1 < l_2$.

□

Le prochain théorème est un résultat très important pour notre étude. Il permet de construire exactement les deuxièmes lignes minimales.

Théorème 3.2 *Une deuxième ligne est minimale si et seulement si elle respecte les lemmes 3.1 et 3.2.*

Preuve

Premièrement : $\Rightarrow$

Il est trivial de prouver ce sens, c'est une application des lemmes.

Deuxièmement : $\Leftarrow$

Le lemme 3.2 dit dans quel ordre écrire les cycles et le lemme 3.1 dit quels éléments y écrire. Comme il n'y a qu'une deuxième ligne minimale pour des cycles donnés, on est certain que notre deuxième ligne est minimale. □

3.4.2 Les rectangles respectant les cycles

Toute cette étude des cycles peut être étendue à un autre résultat important de notre étude. C'est en implantant ce théorème que nous sommes arrivé aux accélérations les plus importantes en termes de temps d'exécution pour la génération des classes d'isomorphisme que nous verrons au chapitre 4. Remarquons d'abord une propriété des cycles de deux carrés latins isomorphes.

Lemme 3.5 *Soit $h : C \rightarrow C'$ un isomorphisme de la boucle C dans la boucle C' et l un ligne quelconque de la table de multiplication de C, alors $\theta(l) = \theta(h(l))$.*

Preuve Pour cette preuve, convenons que $a^k b = a(a^{k-1}b)$ pour $k > 1$ et $a^1 b = ab$.

Prenons un élément a quelconque de C. Soit i un entier tel que $l^i a = a$ dans la boucle C. Alors, il faut que $h(l^i a) = h(a)$ puisque h est une fonction injective. Puisque l'on a posé a quelconque, l'élément $h(l)$ a les mêmes cycles que l. □

Donc, lorsque l'on permute les éléments d'un carré latin, le carré latin isomorphe obtenu a les mêmes cycles. Cette propriété est importante pour prouver le théorème qui suit.

Théorème 3.3 *Soit C un carré latin représentant et l une ligne quelconque de C. Alors $\theta(1) \leq \theta(l)$.*

Preuve Supposons que la ligne l est telle que $\theta(l) < \theta(1)$ dans un carré latin représentant. Alors, il est possible de générer un isomorphe C' de C à l'aide d'une permutation échangeant l pour 1. Il suffit ensuite d'appliquer la bonne permutation ne renommant pas l'élément 1 pour trouver l'isomorphe C'' ayant une deuxième ligne minimale. Comme les cycles de l étaient plus petits que ceux de 1 dans C, on trouve un carré latin C'' plus petit que C, contradiction de l'hypothèse qui affirme que C est représentant. □

3.5 Conclusion

Nous avons exposé dans ce chapitre l'ensemble de la théorie qui est utilisée au chapitre 4 pour générer les classes d'isomorphisme de boucles. Nous y utilisons nos observations sur les cycles pour accélérer l'algorithme en ne générant que des carrés latins ayant des deuxièmes lignes minimales. Nous ne générerons également que des carrés latins dont toutes les lignes ont des cycles plus petits que la deuxième. Nos observations permettent de réduire la difficulté de la génération des classes d'isomorphisme. Ces améliorations ne sont pas seulement des diminutions de constantes, mais de véritables changements d'ordre.

CHAPITRE 4

L'ALGORITHME DE GÉNÉRATION

Le chapitre 4 est le coeur de la présentation de notre travail. Nous appliquons par des algorithmes les lemmes et théorèmes du chapitre 3. Les premiers algorithmes de génération que nous présentons sont relativement simples. Les sections suivantes modifient ces algorithmes pour réduire les calculs. Ces modifications améliorent l'ordre de grandeur du temps d'exécution théorique de l'algorithme alors que celles du chapitre 5 ne permettent que de réduire la constante multiplicative.

Nous commençons donc par un brossage rapide des algorithmes combinatoires. La section 4.1 définit les algorithmes de génération, d'énumération et de recherche (voir [13], [19]). Ensuite, la section 4.2 fournit l'intuition pour l'algorithme de base de génération des carrés latins réduits. Pour ce faire, l'arbre d'exécution de l'algorithme de base est étudié, puis largement utilisé tout au long du document. Certaines des accélérations présentées plus tard permettent de ne pas fouiller l'arbre en entier, en utilisant une technique semblable au branchement et évaluation progressive (*branch and bounds*) [6].

Au début de la section 4.3, nous définissons quelques termes et symboles présents dans les algorithmes qui suivent. Puis nous présentons un algorithme générant tous les carrés latins réduits d'une même taille sans tenir compte des classes d'isomorphisme. Enfin, par le biais de fonctions, nous présentons l'algorithme générant les classes d'isomorphisme. Comme la fonction calculant les classes d'isomorphisme n'est pas triviale, nous détaillons son calcul immédiatement après. La section est close par l'analyse théorique de l'ensemble de ces algorithmes.

Les deux sections suivantes modifient l'algorithme de la section précédente pour l'accélérer. Les deux premières accélérations coupent les branches de l'arbre de la section 4.2, alors que la dernière diminue le travail aux feuilles.

La section 4.4 utilise le théorème 3.3 sur les cycles pour réduire la taille de l'arbre. Par la suite, l'arbre est une fois de plus tronqué à l'aide des rectangles minimaux.

Cette technique consiste à effectuer des comparaisons aux noeuds de l'arbre au lieu des feuilles. Ce sont ces deux accélérations qui s'approchent du branchement et évaluation progressive. Comme l'implémentation des rectangles latins minimaux exige des modifications aux fonctions de la section 4.3, nous détaillons son implémentation. La conclusion est une fois de plus une analyse théorique du temps d'exécution de l'ensemble de l'algorithme.

La dernière section, la section 4.5, réduit les calculs aux feuilles. Seules les feuilles représentant des carrés latins qui ont certaines propriétés sont générées. Nous utilisons les propriétés de ces carrés latins pour réduire les calculs. Cette dernière accélération nécessite quelques définitions et théorèmes. L'implémentation de ces concepts théoriques n'est toutefois pas exposée en détail.

En conclusion, nous analysons l'ensemble des résultats obtenus du côté des temps d'exécution et des résultats empiriques. Certains de ces résultats étant à notre connaissance originaux.

Comme nous voulons générer les classes d'isomorphisme de boucles, le support utilisé n'a aucune importance. Pour simplifier l'écriture, à moins de mentions contraires, le support est toujours les n premiers entiers, 0 représentant l'élément neutre s'il y en a un.

De plus, comme nous utilisons dans cette section que des boucles finies représentées par leur table de multiplication, nous parlons de carrés latins et non de boucles.

4.1 Les algorithmes combinatoires

Avant d'entrer dans le vif du sujet, la génération des classes d'isomorphisme de boucles, nous introduisons le chapitre en situant notre algorithme. Premièrement, il s'agit d'un algorithme combinatoire, puisque les carrés latins, très liés aux boucles, sont des structures combinatoires. Plus de détails sur ces algorithmes sont disponibles

dans [13].

Il existe trois grandes familles d'algorithmes combinatoires classés selon leur dessein. Il y a les algorithmes de génération, d'énumération et de recherche. Les algorithmes de génération construisent toutes les structures combinatoires d'un type donné. Parmi ceux-ci, notons la génération de permutations qui nous intéresse plus particulièrement.

Ensuite, il y a les algorithmes d'énumération qui calculent le nombre de structures différentes d'un type particulier. Ce type d'algorithmes est souvent plus simple que la génération. Prenons pour exemple, le nombre de sous-ensembles de k éléments d'un ensemble de n éléments, un exemple disponible dans [13]. Nous pouvons facilement compter que :

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} \ ,$$

est exactement le nombre de sous-ensembles. Il est plus complexe de générer ces structures combinatoires. Malheureusement, l'énumération est parfois aussi complexe que la génération et on génère toutes les structures combinatoires pour les compter.

La dernière famille d'algorithmes combinatoires est la recherche d'un exemple. Ces algorithmes permettent de trouver au moins une structure combinatoire ayant une propriété. Ces algorithmes passent souvent par la génération des structures combinatoires pour trouver une structure ayant la propriété, mais ce n'est pas toujours le cas. Il est parfois plus rapide de construire directement la structure répondant à la propriété.

Nous utilisons les trois familles d'algorithmes pour générer les classes d'isomorphisme de boucles. En cours de route, nous classerons chacun des algorithmes combinatoires que nous utilisons. Le lecteur trouvera une présentation générale des algorithmes de génération d'objet à un isomorphisme près dans [19].

4.2 L'intuition

La façon la plus simple de générer les carrés latins utilise un algorithme récursif. Nous allons, avant de le présenter, fournir une représentation graphique intuitive qui est par la suite utilisée pour le reste du chapitre. Comme l'algorithme est récursif, on sait que l'on peut visualiser une exécution à l'aide d'un arbre que nous appelons *arbre d'exécution.* Nous renvoyons le lecteur à la figure 4 pour la représentation de l'arbre d'exécution pour $n = 4$.

Définition 4.1 *Un arbre d'exécution permettant de représenter la génération des carrés latins d'une taille donnée est construit de la façon suivante :*

- *La racine de l'arbre est une matrice $n \times n$ contenant les nombres 0 à $n-1$ en ordre lexicographique sur la première ligne et la première colonne.*
- *Chaque noeud de l'arbre a :*
  1. *k enfants, $k < n$ étant le nombre de symboles pouvant prendre place à la case courante ;*
  2. *une case remplie de plus que son père.*
- *Les feuilles ayant une profondeur $(n-1)^2$ sont des carrés latins réduits, les autres sont des branches inutilement explorées.*

Exemple

Prenons le cas d'une génération des carrés latins d'ordre 4. La figure 4 présente l'arbre d'exécution pour $n = 4$. Les croix représentent les branches inutilement explorées. On voit qu'il y a 4 carrés latins réduits d'ordre 4.

Comme on ne s'intéresse pas aux branches inutilement exploitées, il est intéressant de ne pas les dessiner dans un arbre d'exécution. Pour ce faire, nous allons utiliser le théorème 2.1 qui garantit que l'on peut toujours ajouter une ligne à un rectangle latin.

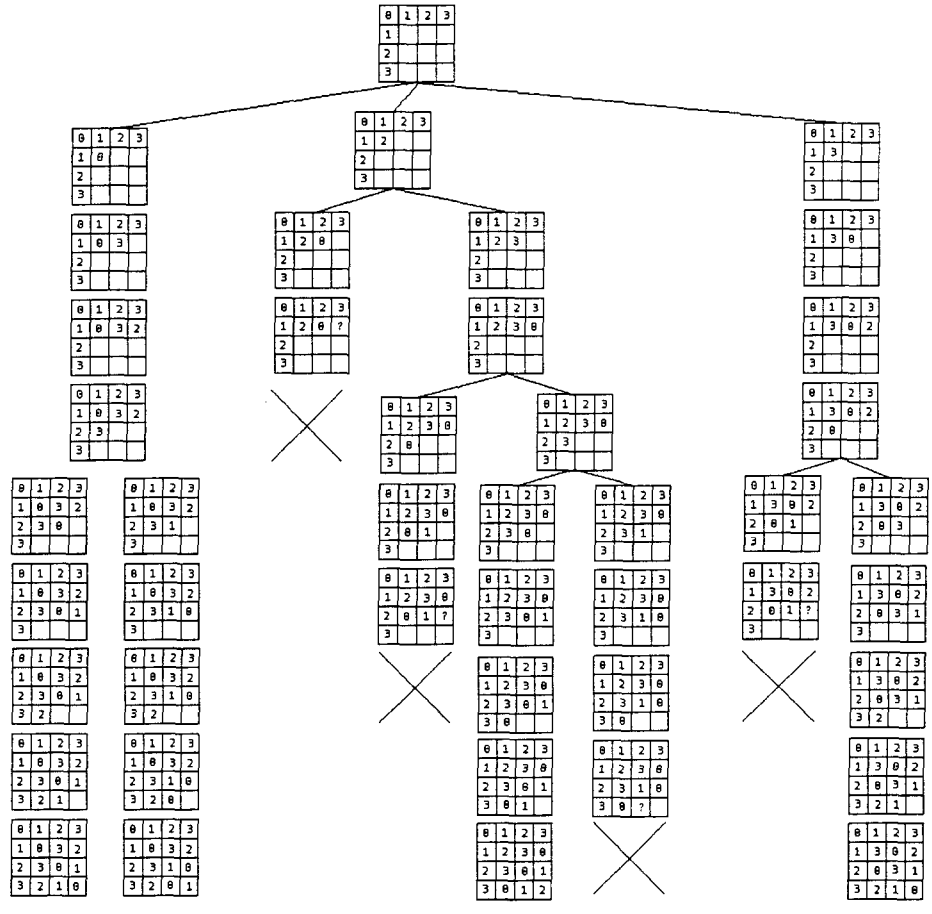

FIG. 4: Arbre d'une exécution pour $n = 4$

Nous représentons ainsi plus simplement un arbre d'exécution. La définition suivante construit l'arbre d'exécution auquel nous faisons référence.

Définition 4.2 *Pour un support B, un* arbre d'exécution *est défini de la façon suivante :*

- *La racine de l'arbre est une matrice $n \times n$ contenant les éléments de B en ordre lexicographique sur sa première ligne et sa première colonne.*
- *Chaque noeud est un rectangle latin ayant pour enfants tous les rectangles latins*

possibles possédant une ligne de plus.

- *Les feuilles sont des carrés latins réduits.*

La figure 5 présente un exemple d'arbre d'exécution pour $n = 4$, $B = \{0, 1, 2, 3\}$ et 0 l'élément neutre.

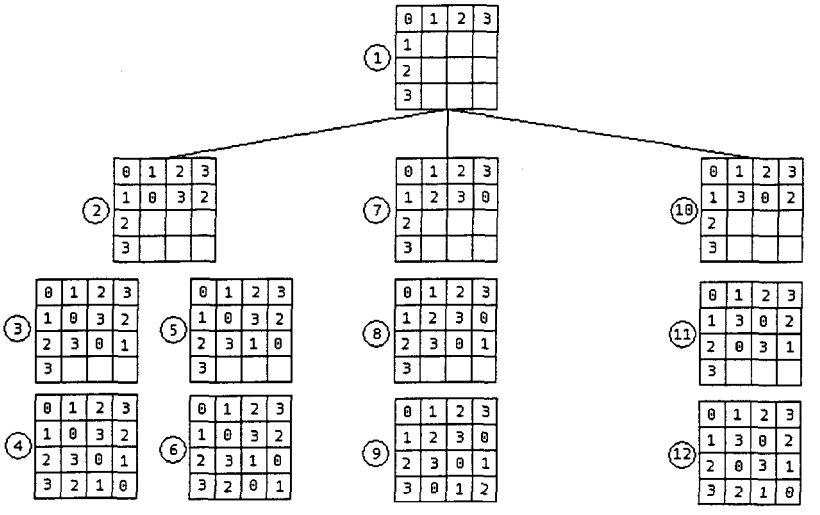

FIG. 5: Arbre d'exécution ($n = 4$)

Nous renvoyons à cette figure plusieurs fois au cours des sections qui suivent. Lorsqu'il est fait mention de branches et de feuilles, c'est à cet arbre qu'il faut se référer ou à un équivalent d'ordre supérieur.

4.3 Algorithme de base

Nous présentons un algorithme qui effectue une fouille en profondeur de l'arbre d'exécution de la section précédente. Ensuite, nous identifions les feuilles de cet arbre illustrant des représentants de classes d'isomorphisme.

Les détails sur l'implémentation de ces calculs sont donnés lorsque l'implémentation

n'est pas évidente. Pour les autres, nous définissons plusieurs fonctions simples pour augmenter la lisibilité et la compréhension des algorithmes.

4.3.1 Définitions

Nous allons maintenant définir les variables qui sont ensuite utilisées dans le chapitre. Nous définissons également certaines fonctions qui servent à la génération des carrés latins. Les méthodes de calcul de ces fonctions peuvent être appelées à être modifiées, les domaines et images restant intacts.

Tout au long du chapitre, nous faisons référence aux définitions suivantes :

Définition 4.3 *B : le support utilisé pour la génération. Nous utilisons, comme c'est généralement le cas pour la génération des carrés latins, les $|B|$ premiers entiers naturels soit $\{0, ..., |B|-1\}$. Puisqu'un élément identité est défini sur B, il sera symbolisé par 0.*

n : cardinalité du support B.

P : un entier compris entre 0 et n^2. Nous allons utiliser un algorithme récursif et P représente la case courante.

C : une matrice $n \times n$ utilisée pour la génération, dont chaque case peut contenir un symbole du support B. C est donc un carré latin partiellement défini.

$\Sigma = B \cup \{\sqcup\}$, où $\sqcup$ est un nouveau symbole. Ce dernier sera utilisé pour indiquer qu'une case d'un carré en construction ne contient pas d'élément de B'.

Avant de présenter quelque algorithme de génération, nous allons définir les fonctions que nous utilisons. Chacune peut être implantée facilement et n'est donc pas détaillée outre mesure.

Les deux premières fonctions permettent, à partir d'une valeur de la position courante P, de trouver la ligne ou la colonne correspondante de la matrice C.

Définition 4.4 *Pour une valeur de n donnée, définissons les deux fonctions suivantes :*

$lig : \mathbb{N} \rightarrow \mathbb{N}$ *tel que* $lig(P) = \lfloor \frac{P}{n} \rfloor$ *où* $\lfloor \frac{P}{n} \rfloor$ *représente la partie entière de* $\frac{P}{n}$.

$col : \mathbb{N} \rightarrow \mathbb{N}$ *tel que* $col(P) = P \bmod(n)$

Ces définitions permettent de vérifier l'identité $lig(P) \times n + col(P) = P$. En effet, $col(P) = P - \lfloor \frac{P}{n} \rfloor \times n$, selon la définition de la division entière et de la division modulo n à partir desquelles on peut facilement calculer l'identité.

La prochaine définition introduit la fonction permettant, lors de la construction, de vérifier si le symbole de la case P est déjà sur la ligne ou sur la colonne. Cela permet de ne générer que des carrés latins (loi d'annulation).

Définition 4.5 $Lat : \Sigma^{n^2} \times \mathbb{N} \rightarrow \{VRAI, FAUX\}$ *est définie de la façon suivante :*

$$Lat(C,P) = \begin{cases} VRAI \text{ si les deux conditions suivantes sont vérifiées :} \\ \quad - \text{ Sur la ligne } lig(P) \text{ de } C \text{ on ne retrouve pas deux fois le même élément de } B \text{ (le symbole } \sqcup \text{ peut apparaître plusieurs fois).} \\ \quad - \text{ Sur la colonne } col(P) \text{ de } C \text{ on ne retrouve pas deux fois le même élément de } B \text{ (le symbole } \sqcup \text{ peut apparaître plusieurs fois).} \\ FAUX \text{ dans les autres cas.} \end{cases}$$

La définition suivante définit dans quel ordre les cases sont remplies lors d'une exécution de l'algorithme. Le calcul de cette fonction est appelé à changer pour permettre de générer des arbres d'exécution différents (voir chapitre 5).

Définition 4.6 *La fonction* $suiv_n : \mathbb{N} \rightarrow \mathbb{N}$ *est définie selon une taille* n *pour* $0 \leq x \leq n$, *de la façon suivante :*

$$suiv_n(x) = \begin{cases} x+1 \text{ si } x+1 < n^2 ; \\ n^2 \text{ sinon.} \end{cases}$$

Lorsqu'il n'y a pas de confusion possible sur la valeur de n nous écrirons seulement *suiv* pour signifier $suiv_n$.

4.3.2 Algorithme de génération de carrés latins

Nous avons maintenant tous les outils nécessaires pour présenter un algorithme générant les carrés latins.

L'algorithme récursif est assez simple. Nous allons tout de même le présenter en détails puisqu'il forme la base sur laquelle les autres algorithmes sont construits. Il se décompose en trois étapes. En analogie avec l'arbre d'exécution de la définition 4.2, la première étape détaille la racine, la seconde les noeuds et la dernière, les feuilles de ce même arbre.

1. Il suffit de commencer le calcul avec une structure ayant les éléments du support B en ordre lexicographique sur sa première ligne et sa première colonne. On commence le calcul à la case $2 \times n + 1$.

2. On crée les enfants de chaque noeud en plaçant dans C, à la position P, tous les éléments de B permettant de respecter les lois d'annulation. Pour chacun, on recommence le même manège avec $P = suiv(P)$.

   Notons que si P est sur la première colonne, aucun calcul n'est effectué. Nous ne voulons que des carrés latins réduits et on passe donc en tel cas à $suiv(P)$.

3. Lorsque $P \geq n^2$, alors le précédent était le dernier élément à remplir (voir definition 4.6). On a donc rempli toutes les cases de C qui est ainsi un carré latin.

Exemple

Voici pour $n = 4$, de quoi pourrait avoir l'air l'ensemble des structures de données à l'étape 1 :

- $B = \{0, 1, 2, 3\}$
- $n = 4$
- $P = 5$
- $C =$

1	0	0	0
2	0	0	0
3	0	0	0

L'algorithme 4.1 qui suit utilise les fonctions des définitions 4.4, 4.5 et 4.6. La méthode naïve pour construire l'arbre d'exécution consiste à ajouter une case à un carré à chaque noeuds pour obtenir ses enfants. Puisque la taille de l'arbre d'exécution est trop grande pour effectuer une fouille en largeur, une fouille en profonfeur est utilisée et implantée par la récursivité. Cet algorithme d'énumération compte le nombre de carrés latins d'une certaine taille.

Notons que, pour ce faire, l'algorithme génère tous les carrés latins. Il suffit d'ajouter, à la ligne 3, une commande pour conserver la structure C avant le retour et on obtient un algorithme de génération. Nous parlons d'ailleurs de la génération de carrés latins même si les algorithmes ne font que compter puisqu'ils peuvent être facilement modifiés pour calculer la génération.

Algorithme : 4.1

```
1   CompteCarre(C, P)
2       SI P ≥ n² ALORS
3           retourne 1
4       SI col(P) = 0 ALORS
5           P = suiv(P)
6       Nombre = 0;
7       POUR chaque x ∈ B FAIRE
8           C[lig(P)][col(P)] = x
9           SI Lat(C, P) ALORS
10              Nombre = Nombre + CompteCarre(C, Suiv(P))
11      RETOURNE Nombre
```

4.3.3 Génération des classes d'isomorphisme

Nous savons qu'il n'y a que 2 classes d'isomorphisme pour les boucles d'ordre 4. Or nous avons 4 carrés latins complétés aux feuilles de l'arbre de la figure 5. Donc, certains sont dans la même classe d'isomorphisme.

L'algorithme de base pour séparer en classes d'isomorphisme ces quatre carrés latins consiste à générer pour chacun le représentant de sa classe. Il faut ensuite vérifier si ce représentant est déjà dans la liste des représentants trouvés. Si c'est le cas, rien n'est fait. Sinon, on vient de trouver un nouveau représentant et donc une nouvelle classe d'isomorphisme, on l'ajoute à une liste.

L'algorithme que nous présentons est un peu plus rapide. Il consiste, à chacune des feuilles de l'arbre d'exécution, à chercher un carré latin isomorphe plus petit et ainsi prouver que le carré latin n'est pas représentant. Si on ne réussit pas, le carré latin est représentant.

L'idée est d'utiliser un algorithme de recherche au lieu d'un algorithme de génération. En effet, pour prouver qu'un carré n'est pas représentant, il suffit de trouver un isomorphe plus petit (recherche). Pour prouver qu'il est le représentant, il faut prouver que tous ses isomorphes sont plus grands ou égaux (génération).

Comme il y a beaucoup plus de carrés latins que de classes d'isomorphisme, le pire cas (un représentant) représente une fraction des feuilles. Le tableau 2 permet de vérifier le nombre de représentants de classes d'isomorphisme par rapport au nombre de carrés latins total. Ajoutons que puisque les carrés latins sont générés en ordre lexicographique, les représentants le sont donc aussi.

Nous représentons cette recherche d'un carré latin plus petit par une fonction qui répond au problème de décision : «Le carré C est-il représentant ?»

Définition 4.7 *Soit la fonction rep* : $B^{n^2} \rightarrow \{VRAI, FAUX\}$, *définie de la façon suivante :*

$$rep(C) = \begin{cases} VRAI \text{ si } C \text{ est un carré latin de taille } n \times n \text{ représentant} \\ \text{de sa classe d'isomorphisme} \\ FAUX \text{ dans les autres cas.} \end{cases}$$

La fonction *rep* est un peu plus complexe à calculer que celles de la section précédente. Nous allons présenter un algorithme qui permet de calculer *rep*, en utilisant quelques autres fonctions qui, elles, sont simples.

4.3.4 Calculer la fonction *rep*

Commençons par définir les fonctions que nous utilisons pour calculer la fonction *rep*. Premièrement, il faut pouvoir générer toutes les permutations. La technique consiste, à partir d'une permutation, à calculer la suivante dans l'ordre lexicographique. Commençons par un exemple pour enchaîner ensuite avec la définition.

Exemple

Voici toutes les permutations fixant l'élément neutre de l'ensemble $\{0,1,2,3,4\}$ *:*

1) 01234	7) 02134	13) 03124	19) 04123
2) 01243	8) 02143	14) 03142	20) 04132
3) 01324	9) 02314	15) 03214	21) 04213
4) 01342	10) 02341	16) 03241	22) 04231
5) 01423	11) 02413	17) 03412	23) 04312
6) 01432	12) 02431	18) 03421	24) 04321

Par exemple une permutation $\pi = 01432$ *a comme suivante* $\pi' = 02134$*. On voit que pour une taille* n *il y a* $(n-1)!$ *permutations. Ces* $(n-1)!$ *permutations permettent de générer tous les isomorphes d'un carré latin.*

Définition 4.8 *Soit* S_n *l'ensemble de toutes les permutations sur* B*. Définissons la fonction* $inc : S_n \rightarrow S_n$*, telle que* $inc(\pi)$ *est la permutation suivante de* π *dans l'ordre lexicographique. Notons que nous considérons que le suivant du dernier élément est lui-même.*

On peut trouver plusieurs algorithmes trouvant le successeur lexicographique d'une permutation dans [13]. Ces algorithmes prennent $O(n^2)$ en pire cas.

Nous allons évidemment devoir calculer des isomorphes pour arriver à évaluer *rep*. La fonction *app* suivante permet de calculer l'isomorphe généré par une permutation π sur un carré C.

Définition 4.9 *Soit* S_n *l'ensemble des permutations sur* B*. Définissons la fonction* $app : \Sigma^{n^2} \times S_n \rightarrow \Sigma^{n^2}$*, telle qu'avec* $C = (B, \cdot)$ *une boucle et* π *une permutation,* $app(C, \pi)$ *est l'unique boucle* C' *telle que* $\pi : C = C'$ *est un isomorphisme.*

Exemple

On pourrait avoir la permutation π et les carrés latins C et C' suivants :

$$C = \begin{array}{|c|c|c|c|c|}\hline 0&1&2&3&4\\\hline 1&3&4&0&2\\\hline 2&0&1&4&3\\\hline 3&4&0&2&1\\\hline 4&2&3&1&0\\\hline\end{array},\ \pi = (0)(1)(32)(4),\ C' = \begin{array}{|c|c|c|c|c|}\hline 0&1&2&3&4\\\hline 1&2&0&4&3\\\hline 2&4&3&0&1\\\hline 3&0&4&1&2\\\hline 4&3&1&2&0\\\hline\end{array}$$

Alors un appel de $app(C, \pi)$ retourne le carré latin C'.

Pour calculer *app*, il suffit de calculer l'élément à mettre dans chaque case de C'. Nous expliquons le calcul en passant par les boucles. Prenons deux boucles $(B, *)$ et $(B, \cdot)$ isomorphes par la fonction π. Alors $\pi(a * b) = \pi(a) \cdot \pi(b)$, ou encore $a * b = \pi^{-1}(\pi(a) \cdot \pi(b))$.

Donc, on peut calculer en temps constant la valeur de chaque élément du carré latin isomorphe. Comme il y a n^2 éléments dans un carré latin, le calcul de *app* est dans $O(n^2)$.

L'algorithme 4.2 qui suit utilise les deux fonctions *inc* et *app* et pour calculer la fonction *rep*. Notons qu'il s'agit d'un algorithme de recherche, nous recherchons un isomorphe plus petit.

Algorithme : 4.2

```
1  rep(C)
2      π = (0, 1, 2, 3, 4, ..., n − 1)
3      FAIRE
4          π = inc(π)
5          SI app(C, π) < C ALORS
6              RETOURNE FAUX
7      TANT QUE π < (0, n − 1, ..., 1)
8      RETOURNE VRAI
```

L'analyse de l'algorithme calculant *rep* nous permet de calculer qu'en pire cas $(n-1)!$ itérations appelant *app* sont faites. De plus, $(n-1)!$ appels de *inc* sont faits toujours en pire cas. Donc, *rep* est dans $O((n-1)! \times (O(app) + O(inc)) = O((n-1)! \times n^2)$. Cet algorithme est amélioré à la section 4.5 de façon significative.

L'algorithme 4.3 suivant permet de calculer le nombre de classes d'isomorphisme en utilisant la fonction *rep*. Puisqu'un représentant est unique, en comptant le nombre de représentants, on compte le nombre de classes. Notons qu'il s'agit toujours d'un algorithme d'énumération.

Il s'agit d'une modification de l'algorithme *CompteCarre* dont la ligne 3 a été remplacée ici par les lignes 3 à 6. Celles-ci permettent de vérifier si le carré latin généré est un représentant. C et P représentent toujours les mêmes choses définies à la section 4.3.

Algorithme : 4.3

```
1   CompteRep(C, P)
2       SI P ≥ n² ALORS
3           SI rep(C) ALORS
4               RETOURNE 1
5           SINON
6               RETOURNE 0
7       SI col(P) = 0 ALORS
8           P = suiv(P)
9       Nombre = 0;
10      POUR chaque x ∈ B FAIRE
11          C[lig(P)][col(P)] = x
12          SI Lat(C, P) ALORS
13              Nombre = Nombre + CompteRep(C, Suiv(P))
14      RETOURNE Nombre
```

4.3.5 Analyse théorique

Nous allons maintenant effectuer une analyse sommaire de l'algorithme 4.3. Nous allons également donner quelques suggestions d'implémentation permettant d'accélérer

l'algorithme.

Premièrement la fonction *inc* n'a pas à être calculée. Pour de petites valeurs de n, il est possible de classer les permutations dans une structure de donnée et ensuite s'y référer en temps constant.

L'algorithme *CompteRep*, si on ne considère pas le coût de la récursion, prend :

$O(rep) + n \times (O(lig) + O(col) + O(Lat)) = O((n-1)! \times n^2)$.

Or il est difficile de prévoir le nombre de fois que *CompteRep* est exécuté. En effet, *CompteRep* est appelé à chaque noeud de l'arbre d'exécution complet. Nous connaissons certaines valeurs, fournies dans le tableau 2 de la page 39, qui permettent d'évaluer le nombre de noeuds de l'arbre à fouiller. Cependant ces valeurs sont à elles seules un sujet d'étude complet.

Nous évaluons donc le temps d'exécution de *CompteRep* en fonction du nombre de carrés latins. Plus loin, nous évaluerons le temps d'autres algorithmes semblables à *CompteRep* en fonction du nombre de classes d'isomorphisme.

Définition 4.10 *Soit carré* : $\mathbb{N} \rightarrow \mathbb{N}$ *tel que carré*(n) *est le nombre de carrés latins réduits d'ordre* n.

Définition 4.11 *Soit iso* : $\mathbb{N} \rightarrow \mathbb{N}$ *tel que iso*(n) *est le nombre de classes d'isomorphisme de carrés latins d'ordre* n.

L'évaluation de ces deux fonctions pour les valeurs connues à ce jour est disponible à la section 2.7 (page 38). Ces valeurs sont tirées de communications privées avec Wendy Myrvold de l'université de Victoria qui les a obtenues de façon indépendante. Nos propres recherches nous ont permis de corroborer le nombre de classes d'isomorphisme pour $n \leq 8$.

Nous savons donc que l'algorithme *CompteRep* prend un temps :

$O(CompteRep) = O(carré(n) \times (n-1)! \times n^2)$

4.4 Amélioration de l'algorithme

Nous allons maintenant utiliser les connaissances du chapitre 3 pour améliorer l'algorithme 4.3. Observons d'abord deux choses sur l'algorithme de base précédent. Premièrement, il utilise $(n-1)!$ permutations en pire cas pour calculer la fonction *rep*. Deuxièmement, il effectue cette opération pour tous les carrés latins. C'est en travaillant sur ces deux points que nous améliorons le temps d'exécution de l'algorithme.

Nous allons commencer par réduire le nombre de feuilles de l'arbre d'exécution en utilisant les résultats de la section 3.4. Nous utilisons deux méthodes. La première consiste à construire des carrés dont les cycles de la deuxième ligne sont plus petits ou égaux aux cycles de toute autre ligne. La seconde consiste à faire certaines comparaisons aux noeuds plutôt qu'aux feuilles, coupant ainsi des branches complètes de l'arbre.

Ensuite, ayant moins de feuilles sur l'arbre d'exécution et ainsi plus d'information sur celles-ci, nous réduisons le nombre de permutations à utiliser pour calculer la fonction *rep*.

4.4.1 Amélioration par les cycles

La première façon de couper l'arbre d'exécution est tirée de la section 3.4.2. On sait par le théorème 3.3 que les cycles de la deuxième ligne d'un carré latin représentant doivent être plus petits ou égaux aux cycles de toute autre ligne. On pourrait arrêter la récursion et ainsi couper une branche de l'arbre d'exécution, lorsqu'un rectangle ne respecte pas la taille des cycles.

L'algorithme 4.4 suivant montre comment effectuer cet arrêt de récursion. Les lignes 8 et 9 calculent les cycles (θ) de la deuxième ligne et ceux de la nouvelle ligne pour les comparer. Si les cycles de la nouvelle sont plus petits que ceux de la deuxième, alors la récursion est arrêtée.

Algorithme : 4.4

```
1   CompteRep2(C, P)
2       SI P ≥ n² ALORS
3           SI rep(C) ALORS
4               RETOURNE 1
5           SINON
6               RETOURNE 0
7       SI col(P) = 0 ALORS
8           SI θ(C[lig(P) − 1]) < θ(C[1]) ALORS
9               RETOURNE 0
10          P = suiv(P)
11      Nombre = 0;
12      POUR chaque x ∈ B FAIRE
13          C[lig(P)][col(P)] = x
14          SI Lat(C, P) ALORS
15              Nombre = Nombre + CompteRep2(C, Suiv(P))
16      RETOURNE Nombre
```

Il est simple de calculer θ à partir d'une représentation en cycle.

Ce calcul se fait dans un temps $O(n^2)$. De plus, il existe plusieurs façons de convertir de la représentation point à point à la représentation en cycle et l'inverse. Nous suggérons [13], pour plus de détails sur ces algorithmes.

Les tableaux 3, 4 et 5 donnent le nombre de noeuds des arbres d'exécution selon leur profondeur. Les deuxième et troisième colonnes indiquent le nombre de noeuds de l'arbre d'exécution construit avec l'algorithme correspondant. La quatrième colonne indique le nombre de noeuds où le calcul des cycles a identifié un rectangle latin ne respectant pas les cycles.

Parce que les algorithmes *CompteRep* et *CompteRep2* sont relativement lents, nous ne disposons des statistiques que pour les ordres de carrés latins plus petits que 8. Déjà sur ces trois tableaux, on peut remarquer que le nombre de feuilles est beaucoup plus petit dans la troisième colonne que dans la deuxième. De l'ordre de la demie pour $n = 5$, du tiers pour $n = 6$ et du quart pour $n = 7$.

$Profondeur$	$CompteRep$	$CompteRep2$	$\theta(lig(P)-1) < \theta(1)$
1	1	1	0
2	11	11	0
3	45	46	12
4	56	44	12
5	56	32	12

TAB. 3: Nombre de noeuds de l'arbre d'exécution, $n = 5$.

$Profondeur$	$CompteRep$	$CompteRep2$	$\theta(lig(P)-1) < \theta(1)$
1	1	1	0
2	53	53	0
3	1064	1064	372
4	6552	4258	1234
5	9408	4276	1044
6	9408	3232	744

TAB. 4: Nombre de noeuds de l'arbre d'exécution, $n = 6$.

4.4.2 Les rectangles minimaux

Nous allons dans la prochaine section couper encore d'autres branches de l'arbre d'exécution. Pour ce faire, nous effectuons aux noeuds de l'arbre d'exécution certaines comparaisons qui sont faites aux feuilles par l'algorithme $CompteRep2$.

Lorsque l'on exécute $CompteRep2$, on effectue $(n-1)!$ comparaisons à chaque feuille en pire cas. Supposons que la permutation $\pi = (12)$ permette de trouver à partir de C, un carré latin C' plus petit. Alors la fonction $rep(C)$ retournera assurément $FAUX$. Supposons maintenant que C' soit plus petit que C sur l'une de ses trois premières

$Profondeur$	$CompteRep$	$CompteRep2$	$\theta(lig(P)-1) < \theta(1)$
1	1	1	0
2	309	309	0
3	35 792	35 792	13 718
4	1 293 216	797 472	246 516
5	11 270 400	4 801 224	1 171 848
6	16 942 080	5 453 640	1 043 556
7	16 942 080	4 410 084	670 068

TAB. 5: Nombre de noeuds de l'arbre d'exécution, $n = 7$.

lignes. Si R et R' sont respectivement les rectangles correspondant aux trois premières lignes de C et C', on peut obtenir R' par une application de la permutation π sur R.

C'est en généralisant ce raisonnement que nous coupons dans l'arbre à certains noeuds, n'explorant pas les sous-arbres. Inspiré de cette idée, nous présentons ce que nous voulons faire, et ensuite nous prouvons que cela nous permet de générer toutes les classes d'isomorphisme.

4.4.3 Implémentation des rectangles latins minimaux

Définition 4.12 *Soit un rectangle latin R de taille $r \times n$. Prenons le carré latin C comme le rectangle latin R dont les $n(n-r)$ cases vides sont remplies avec le symbole $\sqcup$ (vide). Si C est un carré latin minimal, alors on dira que R est un rectangle latin minimal.*

Notons que l'on considère qu'un vide a une valeur supérieure à tout élément du support ($\sqcup > b$, $\forall b \in B$).

Prenons un rectangle latin R de taille $r \times n$, $r \leq n$, en cours de récursion dans

l'algorithme *CompteRep2*. Alors, il est possible de permuter les éléments en respectant certaines règles et ainsi vérifier si le rectangle est minimal. S'il ne l'est pas, il est impossible de le compléter afin d'obtenir un carré latin minimal.

Exemple

Le rectangle R a trois lignes. On peut permuter le 4 *et le* 5 *pour obtenir le rectangle R'.*

$$\begin{array}{cc} & 0\ 1\ 2\ 3\ 4\ 5 \\ R = & 1\ 0\ 3\ 2\ 5\ 4\ , \\ & 2\ 5\ 1\ 4\ 3\ 0 \end{array} \qquad \pi = (54), \qquad \begin{array}{cc} & 0\ 1\ 2\ 3\ 4\ 5 \\ R' = & 1\ 0\ 3\ 2\ 5\ 4 \\ & 2\ 4\ 1\ 5\ 0\ 3 \end{array}$$

Donc tous les enfants de R peuvent être ramenés à un enfant de R' par cette même permutation et R ne peut avoir aucun enfant représentant.

Nous allons introduire une nouvelle notation pour représenter ces permutations. Cette notation permet de définir des familles de permutations plutôt qu'une seule.

Définition 4.13 *Soit un ensemble $E = \{a_1, a_2, ..., a_n\}$ et deux sous-ensembles S et T, $S = \{s_1, s_2, ..., s_k\}$ et $T = \{t_1, t_2, ..., t_l\}$, tel que $S \cap T = \emptyset$. On écrit $[S][T]$ ou $[s_1, s_2, ..., s_k][t_1, t_2, ..., t_l]$ pour représenter l'ensemble des permutations agissant indépendamment sur les éléments de S ainsi que sur ceux de T.*

Exemple

Prenons l'ensemble $E = \{0,1,2,3,4,5\}$, $S = \{1,2\}$ et $T = \{3,4,5\}$. Alors, $[S][T] = [12][345]$ est l'ensemble des permutations suivantes (représentées point à point) :

1) 0 12 345	4) 0 12 453	7) 0 21 345	10) 0 21 453
2) 0 12 354	5) 0 12 534	8) 0 21 354	11) 0 21 534
3) 0 12 435	6) 0 12 543	9) 0 21 435	12) 0 21 543

Mais quelles sont les permutations que l'on peut appliquer à un rectangle latin de r lignes pour le réduire ? Si on prend les n premiers nombres naturels comme support, on peut utiliser toutes les permutations appartenant à $[1, ..., r-1][r, ..., n-1]$.

Les permutations de ce type agissent d'une part sur les lignes qui sont déjà remplies et, de façon indépendante, sur les lignes qui ne le sont pas encore. Pour un rectangle latin $r \times n$, le nombre de permutations de ce type est $(r-1)! \times (n-r)!$.

Nous allons étendre la définition de représentant aux rectangles latins. Cependant, pour éviter toute ambiguïté, nous disons qu'un rectangle est minimal plutôt que représentant.

Définition 4.14 *Un rectangle latin R de taille $r \times n$, $r \leq n$ est dit* minimal *si tous les isomorphes générés par des permutations appartenant à $[1, ..., r-1][r, ..., n-1]$ sont plus grands ou égaux à R.*

Pour implémenter ce nouveau concept, il faut élargir un peu la définition des fonctions *inc*, *rep* et *app* et les généraliser aux rectangles latins. Prenons la fonction *inc*2 suivante qui permet de générer les permutations appartenant à $[a_1, ..., a_{r-1}][a_r, ..., a_{n-1}]$, où les a_i sont les $r-1$ plus petits éléments sans compter l'élément neutre.

Définition 4.15 *Soit $\pi = \pi_1\pi_2$ où $\pi_1 \in [0, ..., r-1]$ et $\pi_2 \in [r, ..., n-1]$, $\pi_0 = (0)$. Soit la fonction $inc2 : B^n \times \mathbb{N} \rightarrow B^n$, définie de la façon suivante.*

$$inc2(\pi, r) = \begin{cases} \pi_1 inc(\pi_2), \text{ si } inc(\pi_2) \neq \pi_2 \\ inc(\pi_1)\pi_0, \text{ sinon.} \end{cases}$$

Enchaînons avec la fonction *app*2 qui fait sensiblement le même calcul que *app*, avec un domaine plus restreint.

Définition 4.16 *Soit la fonction app2* : $B^{r\times n} \times B^n \rightarrow B^{r^2}$ *définie, pour un rectangle* R *de taille* $r \times n$ *et* π *une permutation appartenant à* $[a_1, ..., a_{r-1}][a_r, ..., a_{n-1}]$, *comme la restriction de app aux* r *premières lignes.*

Finalement, la définition de *rep* doit être un peu ajustée :

Définition 4.17 *Soit la fonction rep2* : $\Sigma^{n^2} \times \mathbb{N} \rightarrow \{VRAI, FAUX\}$, *définie de la façon suivante :*

$$rep2(R, r) = \begin{cases} VRAI \text{ si } R \text{ est un rectangle latin minimal de taille } r \times n. \\ FAUX \text{ dans les autres cas.} \end{cases}$$

Théorème 4.1 *Un rectangle latin qui n'est pas minimal ne peut être complété en un carré latin représentant.*

Preuve Supposons qu'un rectangle latin R de taille $r \times n$, $r \leq n$ puisse être réduit au rectangle $R' < R$ par la permutation π. Alors, quelle que soit la façon de compléter R pour obtenir C un carré latin, $app(C, \pi) < C$. □

L'auteur ne connaît qu'un seul cas où une deuxième ligne minimale ne se retrouve chez aucun représentant. En effet, aucun représentant n'a de cycles tel que $\theta(1) = 666665$ pour $n = 6$. Autrement dit, il est impossible de trouver un carré latin d'ordre 6 dont toutes les lignes ont un seul cycle de taille 6.

Implantons ces nouvelles fonctions dans l'algorithme *CompteRep*3. Essentiellement les modifications, en comparaison de l'algorithme *CompteRep*2, sont assez simples. Premièrement, nous avons ajouté la comparaison utilisant *rep*2 à chaque nouvelle ligne, comme pour la comparaison des cycles. Ensuite, puisque cette comparaison pour $P = n^2$ était équivalente à *rep*, nous avons remplacé les quatre premières lignes de l'algorithme par une seule comparaison à la ligne 6.

Algorithme : 4.5

```
1   CompteRep3(C, P)
2       SI col(P) = 0 ALORS
3           SI θ(C[lig(P) − 1]) < θ(C[1]) ALORS RETOURNE 0
4           SI rep2(C, lig(P)) = FAUX ALORS RETOURNE 0
5           P = suiv(P)
6       SI P ≥ n² ALORS RETOURNE 1
7       Nombre = 0
8       POUR chaque x ∈ B FAIRE
9           C[lig(P)][col(P)] = x
10          SI Lat(C, P) ALORS
11              Nombre = Nombre + CompteRep3(C, Suiv(P))
12      RETOURNE Nombre
```

Ce nouvel algorithme permet de couper plusieurs branches de l'arbre. Les branches qui sont coupées ne contiennent, aux feuilles, que des carrés latins en contradiction avec les théorèmes 3.2 ou 3.3.

4.4.4 Analyse

L'algorithme 4.5 permet de réduire le nombre de feuille de l'arbre d'exécution. Nous le croyons, pour $n \leq 8$, dans l'ordre du nombre de classes d'isomorphisme. Nous fournissons à la section 5.3.1 les résultats empiriques qui nous permettent de le croire. Cette amélioration majeure permet de produire des résultats pour $n \leq 8$ dans un temps raisonnable.

Il est une fois de plus difficile d'analyser théoriquement le temps d'exécution de l'algorithme, en fonction de la taille de l'entrée, puisqu'il dépend de l'arbre d'exécution qui n'est connu que pour $n \leq 8$. On peut cependant évaluer le temps d'exécution des fonctions *inc*2, *app*2 et *rep*2.

Commençons par l'analyse de *inc*2. Nous savons que *inc* prend $O(n^2)$ en pire cas. Cependant, comme nous avons des supports de petite taille ($n \leq 9$), il est possible de

garder les $(n-1)!$ permutations dans une structure à accès direct. *inc*2 serait alors dans $O(1)$.

Pour ce qui est de *app*2, on peut considérer qu'il prend un temps du même ordre que *app*. Donc, *rep*2 prend un temps dans le même ordre que *rep* soit $O((n-1)! \times n^2)$. Cependant, comme le montrent les tableaux 6, 7 et 8, l'arbre d'exécution de *rep*2 a un nombre de feuilles dans l'ordre du nombre de classes d'isomorphisme plutôt que dans l'ordre du nombre de carrés latins.

L'ordre de l'algorithme *CompteRep*3 serait donc :

$O(CompteRep3) = O(iso(n) \times (n-1)! \times n^2)$.

L'algorithme *CompteRep*3 exécute deux coupes dans l'arbre d'exécution aux lignes 3 et 4. L'ordre selon lequel on exécute ces coupes ne change pas la taille de l'arbre d'exécution puisqu'il faut que les deux comparaisons soient fausses pour continuer la fouille de l'arbre. Cependant, le calcul de $\theta(lig(P)-1)$ et de $\theta(1)$ est moins long que celui de *rep*2. Donc, il est plus avantageux de couper par les cycles en premier, au niveau du temps réel d'exécution, même si cela ne change pas le temps d'exécution théorique.

Nous allons présenter le nombre de noeuds fouillés pour les ordres 5, 6 et 7 de l'arbre d'exécution de *CompteRep*3 dans les tableaux 6 à 11. Les trois premiers tableaux présentent le nombre de noeuds en coupant avec les cycles en premier, les trois derniers en coupant par les rectangles latins minimaux en premier.

Dans ces tableaux, la première colonne indique la profondeur dans l'arbre d'exécution. Les deux suivantes indiquent le nombre de noeuds, selon la profondeur, des algorithmes *CompteRep* et *CompteRep*3. Les deux dernières colonnes fournissent le nombre de rectangles non minimaux trouvés avec chacune des vérifications, toujours selon la profondeur.

Les trois tableaux 9 à 11 qui suivent présentent le nombre de noeuds si on coupe

$Profondeur$	$CompteRep$	$CompteRep3$	$\theta(lig(P)-1) < \theta(1)$	$rep2\ est\ FAUX$
1	1	1	0	0
2	11	11	0	8
3	46	12	1	2
4	56	11	4	0
5	56	7	1	0

TAB. 6: Noeuds de l'arbre d'exécution, $n = 5$.

$Profondeur$	$CompteRep$	$CompteRep2$	$\theta(lig(P)-1) < \theta(1)$	$rep2\ est\ FAUX$
1	1	1	0	0
2	53	53	0	48
3	1064	99	21	28
4	6552	307	51	0
5	9408	391	54	0
6	9408	337	45	183

TAB. 7: Noeuds de l'arbre d'exécution, $n = 6$.

avec les rectangles latins minimaux avant les cycles. On y compte, comme prévu, le même nombre de noeuds à chaque profondeur que précédemment. Par contre, on compte plus d'exécutions de $rep2$ et moins d'exécutions de $\theta(n-1) < \theta(1)$ pour un arbre ayant les mêmes noeuds. Le temps d'exécution réel était également supérieur.

Cela est consistant puisque le temps d'exécution du calcul de θ est $O(n^2)$ qui est plus grand que celui de $rep2$ qui est $O((n-1)! \times n^2))$.

$Profondeur$	$CompteRep$	$CompteRep2$	$\theta(lig(P)-1) < \theta(1)$	$rep2\ est\ FAUX$
1	1	1	0	0
2	309	309	0	302
3	35 792	808	172	341
4	1 293 216	10 509	2 379	48
5	11 270 400	70 791	11 415	0
6	16 942 080	90 548	11 902	0
7	16 942 080	78 646	8 095	46 805

TAB. 8: Noeuds de l'arbre d'exécution, $n = 7$.

$Profondeur$	$CompteRep$	$CompteRep3$	$rep2\ est\ FAUX$	$\theta(lig(P)-1) < \theta(1)$
1	1	1	0	0
2	11	11	8	0
3	46	12	2	1
4	56	11	0	4
5	56	7	1	0

TAB. 9: Noeuds de l'arbre d'exécution, $n = 5$.

Il semble, après une analyse empirique, que l'algorithme amélioré $CompteRep3$ a un temps d'exécution qui dépend du nombre de classes d'isomorphisme et non du nombre de carrés latins. C'est cette accélération qui nous a permis de croire que nous pourrions générer les classes d'isomorphisme d'ordre 8, chose impossible sans parallélisation par une fouille de l'arbre d'exécution complet.

$Profondeur$	$CompteRep$	$CompteRep2$	$rep2\ est\ FAUX$	$\theta(lig(P)-1) < \theta(1)$
1	1	1	0	0
2	53	53	48	0
3	1064	99	32	17
4	6552	307	0	51
5	9408	391	0	54
6	9408	337	228	0

TAB. 10: Noeuds de l'arbre d'exécution, $n = 6$.

$Profondeur$	$CompteRep$	$CompteRep2$	$rep2\ est\ FAUX$	$\theta(lig(P)-1) < \theta(1)$
1	1	1	0	0
2	309	309	302	0
3	35 792	808	394	119
4	1 293 216	10 509	48	2379
5	11 270 400	70 791	0	11 415
6	16 942 080	90 548	0	11 902
7	16 942 080	78 646	54 900	0

TAB. 11: Noeuds de l'arbre d'exécution, $n = 7$.

4.5 Réduction du nombre de permutations

Maintenant que l'arbre de récursion est bien émondé, nous allons diminuer le travail fait aux noeuds en diminuant l'effort de calcul à chacun. Chaque noeud de l'arbre

d'exécution représente un rectangle latin sur lequel nous avons de l'information. En effet, nous sommes sûr de la vérification de certaines propriétés, la récursion étant arrêtée lorsqu'un rectangle ne s'y conforme pas.

Nous allons d'abord clarifier ces propriétés et fournir les preuves s'y rattachant. Par la suite, nous allons présenter un algorithme d'incrémentation des permutations. Cet algorithme utilise cette information pour réduire le nombre de permutations utilisées afin de déterminer si un carré latin est représentant ou si un rectangle latin est minimal.

4.5.1 Les permutations qui préservent la deuxième ligne

Définition 4.18 *Soit R un rectangle latin de taille $r \times n$, $r \leq n$, et une permutation π. Si la deuxième ligne de app2(R, π) est identique à celle de R, on dit que π* préserve *la deuxième ligne de R.*

Dans cette section, nous allons définir les permutations qui, pour un carré latin C ayant une deuxième ligne minimale, génèrent des carrés latins isomorphes ayant cette même deuxième ligne. Nous procédons en commençant par les permutations qui fixent le 1 et ensuite nous généralisons à toutes les permutations.

Lemme 4.1 *Soit un rectangle latin R produit par l'algorithme 4.5 et π une permutation qui fixe le 1 et préserve la deuxième ligne de R.*

Avec $(B, +)$ et $(B, \times)$ les boucles partiellement définies correspondant respectivement à R et app2(R, π), on a :

1. *si $\pi(a) = b$ alors $\pi(1 + a) = 1 + b$;*
2. *si $\pi(a) = b$ alors le cycle auquel appartient a est de la même taille que le cycle auquel appartient b sur la deuxième ligne de R.*

Preuve

1. Puisque π est un isomorphisme alors :

   $\pi(1+a) = \pi(1) \times \pi(a) = 1 \times b = 1 + b$, puique R et $app2(R, \pi)$ ont la même deuxième ligne.

2. Posons les éléments du cycle de a_0 et de b_0 tels que $\pi(a_0) = b_0$. Supposons que l'on a :

$$\begin{array}{ll} 1 + a_0 = a_1 & 1 \times b_0 = b_1 \\ 1 + a_1 = a_2 & 1 \times b_1 = b_2 \\ 1 + a_2 = a_3 & 1 \times b_2 = b_3 \\ \cdots & \\ 1 + a_{n-1} = a_0 & 1 \times b_{m-1} = b_0 \end{array}$$

   On peut, sans perdre de généralité, supposer $n \leq m$. Nous avons $\pi(a_0) = b_0$ et ainsi :

$$\pi(a_1) = \pi(1 + a_0) = \pi(1) \times \pi(a_0) = 1 \times b_0 = b_1$$

   On peut calculer que $\pi(a_i) = b_i$ pour $0 < i < n$. Cela implique que $\pi(a_{n-1}) = b_{n-1}$. Or $1 + a_{n-1} = a_0$ et donc :

$$\pi(1 + a_{n-1}) = \pi(1) \times \pi(a_{n-1}) = 1 \times b_{n-1} = \pi(a_0) = b_0$$

   Ce qui implique que $m = n$.

□

Donc, si l'on veut préserver la deuxième ligne d'un rectangle latin, on ne peut permuter que des éléments appartenant à des cycles de même taille (sur la deuxième ligne du rectangle). De plus, une fois l'élément fixé, les autres éléments du même cycle le sont aussi.

Exemple

Pour les enfants du rectangle latin minimal suivant :

$$R = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & 2 & 0 & 4 & 3 & 6 & 7 & 5 \\ \hline \end{array}$$

les permutations permettant de générer des rectangles minimaux sont :

1 : 0 1 2 3 4 5 6 7

2 : 0 1 2 3 4 6 7 5

3 : 0 1 2 3 4 7 5 6

4 : 0 1 2 4 3 5 6 7

5 : 0 1 2 4 3 6 7 5

6 : 0 1 2 4 3 7 5 6

Il faut remarquer que l'on ne peut pas permuter les éléments du premier cycle puisqu'il faudrait ainsi renommer le 1. Or, on a supposé que l'on ne renomme pas le 1.

Voyons maintenant les permutations pouvant renommer le 1 qui préservent la deuxième ligne.

Lemme 4.2 *Si une permutation π préserve la deuxième ligne d'un rectangle R généré par l'algorithme 4.5, alors les cycles de l'élément l, tel que $\pi(l) = 1$, sont les mêmes que ceux de* 1.

Preuve Comme R a été généré par l'algorithme 4.5 qui respecte les cycles, nous savons que $\theta(1) \leq \theta(\pi(1))$. Supposons que $\theta(1) < \theta(\pi(1))$, alors $\theta(1)$ sur $app2(R, \pi)$

n'est pas $\theta(\pi(1))$ sur R. Par le lemme 3.4, les deuxièmes lignes de R et R' sont différentes, d'où une contradiction. □

Les deux lemmes 4.1 et 4.2 permettent de réduire grandement le nombre de permutations à utiliser pour déterminer si un rectangle latin est minimal. En effet, on sait que, dans l'algorithme 4.5, les carrés latins générés ont la même deuxième ligne que leur représentant.

La prochaine section détaille l'implémentation qui permet de fusionner les lemmes 4.1 et 4.2.

4.5.2 La génération des permutations qui préservent la deuxième ligne

Les lemmes de la section précédente permettent de réduire grandement le nombre de permutations à considérer pour identifier les représentants de classes d'isomorphisme. Nous allons évaluer le nombre des permutations qui préservent la deuxième ligne au chapitre 5. Nous nous intéressons ici à un algorithme pour générer ces permutations.

Définition 4.19 *Pour un rectangle latin minimal R, on appelle* permutations réduites *l'ensemble des permutations préservant la deuxième ligne de R.*

Puisqu'il est difficile de générer les permutations réduites, nous allons décomposer les permutations préservant la deuxième ligne en un produit de trois permutations. Ces trois permutations ont l'avantage d'appliquer directement les lemmes 4.1 et 4.2 de la section précédente et d'être plus faciles à calculer.

Définition 4.20 *Soit R un rectangle latin minimal et une permutation π qui préserve la deuxième ligne de R. Alors il est possible de calculer π_1, π_2, π_3 trois permutations satisfaisant les conditions suivantes :*

1. $\pi = \pi_3\ \pi_2\ \pi_1$

2. π_1 *échange* 1 *et* l, $\theta(1) = \theta(l)$ *dans* R,

3. $\pi_2 = f(\pi_1)$ *est telle que* $\pi_2 \circ \pi_1$ *préserve la deuxième ligne de* R,

4. π_3 *est de la forme* $[2, 3, ..., n-1]$ *et préserve la deuxième ligne de* $app(R, \pi_2 \circ \pi_1)$.

Nous utilisons ces trois permutations parce qu'il est facile de générer les permutations π_1 *qui sont une utilisation directe du lemme 4.2. Pour ce qui est des permutations* π_3 *qui utilisent le lemme 4.1 la génération est assez aisée. Finalement les permutations* π_2 *sont assez faciles à calculer en* $O(n)$ *à l'aide de l'algorithme 4.6 suivant. Par contre, il est difficile de générer toutes les permutations* π *préservant la deuxième ligne. Or ces mêmes permutations sont exactement le produit de toutes les combinaisons possibles de* π_1, π_2 *et* π_3.

Algorithme : 4.6

```
1  reduire(l)
2      l_0 est la deuxième ligne minimale telle que θ(l_0) = θ(l)
3      π est la permutation identité
4      POUR x allant de 0 à n − 1
5          a = l[x]
6          b = l_0[x]
7          π = ππ' où π' = (ab)
8          l = π'(l)
9      RETOURNE π
```

Définition 4.21 *Pour un rectangle latin minimal* R, *soit* π_1, π_2, π_3 *trois permutations et* $\pi = \pi_3 \circ \pi_2 \circ \pi_1$. *Alors* π *est une permutations réduites si et seulement s'il existe* π_1, π_2, π_3 *répondant à la définition 4.20.*

Exemple

Prenons le rectangle latin suivant :

$$R = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&1&2&3&4&5&6&7\\\hline 1&2&0&4&3&6&7&5\\\hline 2&0&1&6&5&7&3&4\\\hline\end{array}$$

Sur R, on a $\theta(1) = \theta(2) = 38778888$ (voir 3.2.3 pour la définition de θ), prenons $\pi_1 = (12)$. On calcule ainsi que $R_1 = app2(R, \pi_1)$ suivant :

$$R_1 = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&1&2&3&4&5&6&7\\\hline 1&2&0&6&5&7&3&4\\\hline 2&0&1&4&3&6&7&5\\\hline\end{array}$$

La deuxième ligne de R_1 n'est pas minimale et on cherche la permutation π_2 qui la réduira. On utilise donc l'algorithme 4.6 avec l_0 la deuxième ligne de R et l la deuxième ligne de R_1. Notons d'abord que si $l[x] = l_0[x]$, on calcule π' la permutation identité à la ligne 7 de l'algorithme comme c'est le cas avec $0 \leq x \leq 2$ sur les rectangles R et R_1.

Avec x valant 3, on obtient la permutation $\pi' = (l[3](l_0[3]) = (64)$ dans l'algorithme à la ligne 7. On calcule ainsi le rectangle latin $R'_1 = app(R_1, (64))$. L'algorithme 4.6 ne calcule que la deuxième ligne de R'_1 pour un meilleur temps d'exécution, mais pour la compréhension nous présentons ici le rectangle latin complet.

$$R'_1 = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&1&2&3&4&5&6&7\\\hline 1&2&0&4&3&7&5&6\\\hline 2&0&1&6&7&4&3&5\\\hline\end{array}$$

On calcule ensuite la permutation $\pi' = (l[5]l_0[5]) = (67)$ *avec* x *vallant* 5. *On obtient ainsi le rectangle* $R_2 = app(R_1', (67))$ *suivant :*

$$R_2 = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&1&2&3&4&5&6&7\\\hline 1&2&0&4&3&6&7&5\\\hline 2&0&1&7&6&4&5&3\\\hline\end{array}$$

Remarquons que l'on a utilisé successivement les permutations (64) *puis* (76). *Hors* $\pi = (76)(64) = (647)$ *est la permutation calculée à l'aide de l'algorithme reduire qui rend la deuxième ligne de* R_1 *minimale.*

Finalement, l'ensemble des permutations π_3 *de* R_2 *est le produit de toutes les combinaisons possibles des permutations* (0), (43) *avec les permutations* (0), (567), (576).

Il suffit donc, dans le calcul de *inc*2, de calculer à chaque itération si la permutation respecte les lemmes et de passer à la suivante dans le cas contraire.

On peut dénombrer le nombre de permutations π_3 de la définition 4.20. Il suffit, pour un rectangle latin minimal, d'évaluer $\theta(1)$ et d'effectuer le calcul suivant :

Théorème 4.2 *Soit* $\theta(1) = a_0a_1...a_{n-1}$, *sur un rectangle latin minimal. Le nombre de permutations préservant la deuxième ligne sans renommer le* 1 *est :*
$\prod_{i=1}^{n-1} i^{n-a_i} \cdot (n-a_i)!$, *pour* $2 < i < n$.

Preuve Chaque élément de θ représente une taille de cycle. Prenons a_i un élément de θ. Alors, il y a a_i cycles de taille i. Alors il y a $i \times a_i$ candidats pour le premier élément du premier cycle. Il y a un seul candidat pour les autres éléments du cycle. Ensuite, il y a $i \times (a_i - 1)$ éléments pour le premier élément du deuxième cycle. Le manège se répète jusqu'à $i \times 1$. Donc, il y a $i^{a_i} \cdot (a_i)!$ possibilités. Ce calcul doit être fait pour chaque élément de θ.

□

Notons que plus i est grand, moins a_i peut l'être. Donc, le nombre de permutations préservant la deuxième ligne est plus petit que le nombre de permutations totales. Nous voyons au chapitre 4, à la section 5.3.1, le nombre de permutations préservant la deuxième ligne pour chaque deuxième ligne minimale.

4.6 Conclusion

Nous avons, pour résumer, réduit la fouille de l'arbre d'exécution en arrêtant la fouille en profondeur lorsque certaines propriétés n'étaient pas respectées. À un noeud, nous savions si la fouille du sous-arbre pouvait être fructueuse. Ensuite, nous avons réduit le calcul nécessaire pour identifier les représentants de classe d'isomorphisme.

En effet, lorsque certaines propriétés devaient être respectées, nous avions une information précieuse sur les feuilles de l'arbre de récursion. C'est à l'aide de cette information que nous avons grandement diminué l'effort de calcul.

Nous avons vu que, pour des ordres inférieurs à 8, le temps d'exécution de l'algorithme devient fonction du nombre de classes d'isomorphisme plutôt que du nombre de carrés latins. Il s'agit là d'une accélération appréciable qui porte à croire que les efforts pour émonder l'arbre ne pourrait pas apporter encore de grandes accélérations.

Nous avons cependant trouvé empiriquement d'autres accélérations. Elles font l'objet d'une étude au chapitre 5. C'est parce que ces accélérations sont également importantes que nous n'avons pas de mesures pour $n = 8$. Toutes les mesures intéressantes pour cet ordre se retrouvent aussi au chapitre 5.

Nous sommes tout de même arrivé à générer, avec une implémentation de l'algorithme *CompteRep*3 utilisant les permutations réduites, toutes les classes d'isomorphisme de boucles d'ordre 8. À notre connaissance, il s'agissait là d'un résultat original. Cela nous a également permis de corroborer les résultats de Wendy Myrvold énumérant ces classes à la valeur de 106 228 849 de façon indépendante. L'exécution du

programme a mis huit heures sur une machine possédant un processeur de 900 MHz.

CHAPITRE 5

ANALYSE EMPIRIQUE

En effectuant une recherche empirique, nous avons accéléré notre algorithme et cueilli des données sur l'arbre d'exécution. Ce chapitre se consacre aux plus importantes des ces accélérations et données.

D'abord, à la section 5.1 nous présentons les accélérations que nous avons trouvées empiriquement. Celles-ci se rapportent au calcul de la loi d'annulation par la fonction *Lat* (section 5.1.1), au calcul de la fonction θ servant à calculer les cycles des permutations (5.1.2) et finalement à l'essai de nouvelles relations d'ordre de boucles(5.1.3).

En conclusion, un calcul permettant d'estimer le temps réel que demande une exécution pour $n = 9$ est présenté.

5.1 Accélérations empiriques

Nous avons construit l'algorithme *CompteRep*3 du chapitre précédent à partir de l'algorithme de base *CompteRep* grâce à une étude théorique du problème. Une étude, partiellement empirique cette fois, nous a permis de réduire encore le temps d'exécution.

Certains des calculs faits par l'algorithme *CompteRep*3 sont redondants si on effectue une implémentation simple de l'algorithme théorique et une programmation bête des fonctions. La présente section a pour objectif de présenter les ajustements en implémentation qui, sans changer l'ordre d'exécution de l'algorithme, améliorent de façon appréciable la constante multiplicative.

Le premier ajustement se fait au niveau du calcul de la fonction *Lat* (voir définition 4.5). L'algorithme *CompteRep*3 de la page 81, à chaque appel, effectue n appels de *Lat*. Chacun de ces appels prend $2n$ itérations en pire cas. Entre ces appels, une seule case de la structure C est modifiée et les mêmes calculs sont faits par la fonction *Lat*. En fait, un seul appel est nécessaire et il suffit de préserver l'information calculée par *Lat*, pour ensuite s'y référer en temps constant.

Ensuite, il y a les calculs de la fonction θ (voir section 3.5) permettant de comparer les cycles de deux permutations. Cette fonction est appelée deux fois à chaque itération de l'algorithme *CompteRep*3 (voir section 3.2.3). Une fois pour la nouvelle ligne construite et une seconde pour la deuxième ligne. Nous savons (selon l'arbre d'exécution) que le nombre de deuxièmes lignes minimales est beaucoup plus petit que le nombre total de noeuds. Il y a donc des calculs redondants.

Troisièmement, il y a la relation d'ordre des boucles (voir section 3.2.2) qui peut être modifiée. La relation d'ordre définit comment les cases du carré latin C sont remplies dans l'algorithme. En changeant cette relation d'ordre, nous générons les carrés latins dans un ordre différent et nous coupons ainsi un arbre d'exécution différent.

Ces accélérations, qui sont détaillées dans les prochaines sections, ont permis de réduire significativement le temps d'exécution. Sur une machine à 450 MHz, notre solution accélérée sur une entrée de taille $n = 8$ a répondu en 6 heures comparativement à une semaine précédemment.

5.1.1 Calcul de la loi d'annulation

La première amélioration empirique a été trouvée en évaluant la distribution du temps d'exécution entres les procédures, sur une implémentation basée sur l'algorithme *CompteRep*3. Nous nous sommes ainsi aperçu qu'un module prenait plus du quart du temps d'exécution total, le module implantant la fonction *Lat*.

Il est simple de comprendre que plusieurs calculs faits par *Lat* sont redondants. À chaque appel de *CompteRep*3, la fonction *Lat* est appelée n fois. À chacun de ces appels, C n'est modifié qu'en une seule valeur et P reste constant.

Il serait facile d'identifier une seule fois par appel de *CompteRep*3 les éléments de la colonne et de la ligne de la case courante qui sont libres et garder cette information dans une structure de données à accès direct. Il serait ensuite possible de vérifier en

temps constant si une valeur est libre.

Nous utilisons un tableau pour garder le résultat du calcul des cases libres. Ensuite, il suffit de vérifier quelles valeurs peuvent être mises à la case P pour respecter les lois d'annulation. Nous avons divisé en deux fonctions ce calcul.

Définition 5.1 *Soit la fonction lib* : $\Sigma^{n^2} \times \mathbb{N} \times \Sigma \rightarrow \{VRAI, FAUX\}$, *définie comme suit :*

$$lib(C, P, s) = \begin{cases} VRAI \text{ si la colonne } col(P) \text{ et la ligne } lig(P) \text{ de } C \text{ ne} \\ \text{contiennent pas le symbole } s. \\ FAUX \text{ dans les autres cas.} \end{cases}$$

Définition 5.2 *Soit* $LIB \in \{VRAI, FAUX\}^n$ *le vecteur défini de la façon suivante :*

$$LIB = (lib(C, P, 0), ..., lib(C, P, n-1))$$

Le calcul de LIB demande donc n appels de *lib* prenant n itérations, si on effectue une implémentation bête. Considérons plutôt l'algorithme suivant qui calcule LIB en temps $O(n)$.

Algorithme : 5.1

```
1  CalculLIB(C, P)
2      LIB = VRAI^n
3      POUR x va de 0 a lig(P) FAIRE
4          LIB[C[x][col(P)]] = FAUX
5      POUR x va de 0 a col(P) FAIRE
6          LIB(C[lig(P)][x]) = FAUX
6  RETOURNE LIB
```

L'algorithme *CompteRep*4 qui suit est basé sur l'algorithme *CompteRep*3 pour utiliser le vecteur LIB. La ligne 7 a été ajoutée et la comparaison de la ligne 12 a été ajustée.

Algorithme : 5.2

```
1   CompteRep4(C, P)
2       SI col(P) = 0 ALORS
3           SI θ(C[lig(P) − 1]) < θ(C[1]) ALORS RETOURNE 0
4           SI rep2(C, l(P)) = FAUX  ALORS RETOURNE 0
5           P = suiv(P)
6       SI P ≥ n² ALORS RETOURNE 1
7       LIB = CalculLIB(C, P)
8       Nombre = 0
9       POUR chaque x ∈ B FAIRE
10          SI LIB[x] = VRAI ALORS
12              C[l(P)][c(P)] = x
13              Nombre = Nombre + CompteRep4(C, Suiv(P))
14      RETOURNE Nombre
```

Analysons ce nouvel algorithme comparativement à *CompteRep*3. *CompteRep*3 effectue n appels de la fonction *Lat* qui demandent un temps $O(n)$ en pire cas. Le nouvel algorithme n'effectue qu'un appel de LIB qui demande aussi un temps $O(n)$ en pire cas. Si on considère le nombre de fois que le module *CompteRep*4 est appelé en une exécution ($O(iso(n))$), il s'agit là d'un gain appréciable. De l'ordre de 30 pour cent lors de l'implémentation que nous avons effectuée.

5.1.2 Les cycles de la deuxième ligne

Au cours de cette section, nous allons éliminer des calculs redondants de la fonction θ (voir section 3.2.3). Rappelons que la fonction θ a pour domaine l'ensemble des permutations d'un ensemble. La cardinalité de cet ensemble étant, pour un support S, $n!$ où $n = |S|$.

À chaque noeud de l'arbre d'exécution la fonction θ est calculée pour deux entrées : la permutation représentant la deuxième ligne et une autre permutation représentant la nouvelle ligne. Le nombre de noeuds d'un arbre d'exécution pour n suffisamment grand ($n \geq 7$) est largement supérieur à $n!$. Plusieurs calculs sont donc redondants pour ce qui est de la fonction θ.

La redondance la plus évidente apparaît lors du calcul de la fonction θ pour la deuxième ligne. Sur l'arbre d'exécution, tous les noeuds de profondeur 1 ont des descendants ayant la même deuxième ligne qu'eux. Le calcul de θ n'a donc à être fait qu'une fois pour chacune de ces branches. Il existe plusieurs façons simples d'implémenter cela à partir de l'algorithme *CompteRep*4.

La méthode la plus efficace à laquelle nous pensons serait de calculer la fonction θ sur l'ensemble de son domaine et d'accéder ensuite à l'information en temps constant. Nous avons utilisé une autre méthode plus près de l'arbre d'exécution et plus simple à implémenter.

Cette méthode consiste à séparer la fouille de l'arbre d'exécution en deux algorithmes. Le premier, que nous appelons *CompteRec*, fouille les noeuds de surface dans l'arbre d'exécution. À partir d'une profondeur déterminée, un autre algorithme, semblable à *CompteRep*4, fouille les sous-arbres correspondant aux feuilles de *CompteRec*.

L'avantage de cette méthode est qu'elle permet d'effectuer des calculs différents selon la profondeur de l'arbre. Lors d'expérimentations, cela s'est avéré très utile. Nous avons alternativement utilisé les différentes accélérations pour en constater l'importance aux différentes profondeurs. De plus ce nouvel algorithme est plus facile à paralléliser.

Prenons l'algorithme *CompteRec* suivant qui permet d'effectuer les calculs aux noeuds de surface dans l'arbre d'exécution complet. Il appelle *CompteRep*5 à la ligne 6 pour fouiller les sous-arbres. F est le nombre de cases remplies avant l'appel de *CompteRep*5 pour la fouille du sous-arbre.

L'algorithme *CompteRep*5 est identique à *CompteRep*4 à l'exception de la ligne 3. Cette ligne utilise le résultat du calcul de $\theta(C[1])$ reçu en paramètre dans la variable $L2$ au lieu d'effectuer le calcul.

Algorithme : 5.3

```
1   CompteRec(C, P)
2       SI col(P) = 0 ALORS
3           SI rep2(C, l(P) = FAUX) ALORS RETOURNE 0
4           P = suiv(P)
5       SI P ≥ F ALORS
6           RETOURNE CompteRep5(C, P, θ(C[1]))
7       LIB = CalculLIB(C, P)
8       Nombre = 0
9       POUR chaque x ∈ B FAIRE
10          SI LIB[x] = VRAI ALORS
12              C[l(P)][c(P)] = x
13              Nombre = Nombre + CompteRec(C, Suiv(P))
14      RETOURNE Nombre
```

Algorithme : 5.4

```
1   CompteRep5(C, P, L2)
2       SI col(P) = 0 ALORS
3           SI θ(C[lig(P) − 1]) < L2) ALORS RETOURNE 0
4           SI rep2(C, lig(P)) = FAUX  ALORS RETOURNE 0
5           P = suiv(P)
6       SI P ≥ n² ALORS RETOURNE 1
7       LIB = CalculLIB(C, P)
8       Nombre = 0
9       POUR chaque x ∈ B FAIRE
10          SI LIB[x] = VRAI ALORS
12              C[l(P)][c(P)] = x
13              Nombre = Nombre + CompteRep4(C, Suiv(P))
14      RETOURNE Nombre
```

L'élimination des calculs de cycles redondants a permis de réduire le temps d'exécution de la solution implantée de 20 pour cent pour $n = 7$. Toutefois, aucun test pour $n = 8$ n'a été fait et il est possible que pour cette taille, le gain en temps soit supérieur comme inférieur. En effet, à ce stade l'algorithme $CompteRec(C, P)$ calcule les classes d'isomorphisme pour $n = 7$ en 300 secondes sur un Pentium cadencé à 450 MHz. Nous croyons qu'il faudrait cependant plusieurs semaines pour une exécution de $n = 8$, d'après les essais sommaires que nous avons faits.

5.1.3 Les relations d'ordre

Nous avons défini à la section 3.2.2 une relation d'ordre générale pour comparer les boucles. Nous avions alors signalé que cette relation d'ordre était appelée à changer. Nous allons au cours de cette section poser de nouvelles relations d'ordre de façon à ordonner les cases des boucles de façon différente. Cela permet de construire des arbres d'exécution différents.

Nous allons commencer par définir une matrice, que nous appelons *canevas*, ordonnant les cases des tables de multiplication des boucles.

Exemple

Pour comparer les deux boucles suivantes :

$$B = \begin{array}{c|cccc} + & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 1 & 2 & 3 & 0 \\ 2 & 2 & 3 & 0 & 1 \\ 3 & 3 & 0 & 1 & 2 \end{array} \qquad B' = \begin{array}{c|cccc} \cdot & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 1 & 0 & 3 & 2 \\ 2 & 2 & 3 & 0 & 1 \\ 3 & 3 & 2 & 1 & 0 \end{array}$$

Nous allons ordonner les cases des tables de multiplication de ces deux boucles à l'aide de la matrice M suivante :

$$M = \begin{vmatrix} 0 & 1 & 2 & 3 \\ 4 & 8 & 7 & 9 \\ 5 & 10 & 12 & 13 \\ 6 & 11 & 14 & 15 \end{vmatrix}$$

Ainsi, on compare les cases des tables de multiplication des boucles B et B' selon l'ordre définit par M. On remarque ainsi que les 7 premiers éléments définit par M

sont identiques dans B et B'. Par contre, l'élément 8 est plus grand pour la boucle B (2) que la boucle B' (0).

Définitions

Nous allons maintenant fournir quelques définitions afin de formaliser la notion de canevas dont nous avons besoin pour définir de nouvelles relations d'ordre sur les boucles.

Définition 5.3 *Définissons un* canevas *d'ordre n comme une matrice de taille $n \times n$ remplie avec les nombres 0 à $n^2 - 1$ d'une manière quelconque.*

Un canevas permet d'assigner un numéro unique à chacune des cases de la table de multiplication d'une boucle. Nous appelons canevas standard un canevas de taille n où à la ligne $0 \leq l < n$ et la colonne $0 \leq c < n$ on retrouve la valeur $l \times n + c$. Notons que l'utilisation d'un canevas standard M revient à l'utilisation de la relation d'ordre de la section 3.2.2.

La fonction $f_{(B,M)}$ qui suit associe chaque élément du canevas M à un élément de la boucle B. Ensuite, la fonction Ω_M construit pour chaque boucle le mot qui est comparé en concaténant les n^2 éléments calculés par la fonction $f_{(B,M)}$.

Définition 5.4 *Soit la fonction $f_{(B,M)} : \{0, ..., |B|^2 - 1\} \rightarrow B$ définie, pour $(B, \cdot)$ une boucle et M un canevas, de la façon suivante :*

$$f_{(B,M)}(i) = \begin{cases} \textit{L'élément de la table de multiplication de } B \textit{ à la position correspondant à } i \textit{ dans } M. \end{cases}$$

Exemple

Si on considère le canevas M et la boucle B suivante :

$$M = \begin{vmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{vmatrix} \qquad B = \begin{array}{c|cccc} \cdot & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 1 & 0 & 3 & 2 \\ 2 & 2 & 3 & 0 & 1 \\ 3 & 3 & 2 & 1 & 0 \end{array}$$

Alors $f_{(M,B)}(5) = 0$, *et* $f_{(M,B)}(14) = 1$.

Définition 5.5 *Soit β_i l'ensemble de toutes les boucles de taille i.*

Définition 5.6 *Soit B^n, $n > 0$, l'ensemble des mots de longueur n appartenant au monoïde libre B^*.*

Définition 5.7 *Soit $\Omega_M : \beta_n \rightarrow B^{n^2}$ une fonction définie pour M, un canevas d'ordre n, comme la concaténation des n^2 éléments calculés par $f_{(B,M)}(i)$, $0 \leq i < n^2$.*

Pour comparer deux boucles B et B', nous comparerons lexicographiquement les mots $\Omega_M(B)$ et $\Omega_M(B')$. Lorsqu'il n'y a pas de confusion possible, nous écrivons $B \leq B'$ afin d'écourter $\Omega_M(B) \leq \Omega_M(B')$.

Exemple

Soit M le canevas suivant :

$$M = \begin{vmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{vmatrix}$$

Alors pour la boucle B suivante :

$$B = \begin{array}{c|cccc} \cdot & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 1 & 0 & 3 & 2 \\ 2 & 2 & 3 & 0 & 1 \\ 3 & 3 & 2 & 1 & 0 \end{array}$$

nous avons $\Omega_M(B) = 0123103223013210$.

Implantations

Pour utiliser les canevas dans nos algorithmes nous allons devoir modifier la fonction $suiv_n$, de la définition 4.6 page 44, pour implanter la nouvelle relation d'ordre. Les deux prochaines fonctions permettent de définir la relation d'ordre utilisée pour la construction et la comparaison des carrés latins. La première fonction sert à simplifier l'utilisation du canevas.

Définition 5.8 *Pour* $n \geq 1$ *et* M *un canevas d'ordre* n, *définissons la fonction* can_M : $\{0, ..., n^2\} \rightarrow \{0, ..., n^2\}$ *de la façon suivante :* $can_M(x) = M[lig(x)][col(x)]$.

On voit que can_M est une bijection et il est donc possible d'utiliser can_M^{-1} la fonction inverse. La prochaine fonction permet de calculer, à partir d'une position donnée, la case suivante en respectant le canevas.

$$suiv_M(x) = \begin{cases} can_M^{-1}(can_M(x) + 1) \text{ si } can_M(x) < n^2 - 1\,; \\ n^2 \text{ sinon.} \end{cases}$$

Exemple

Pour bien comprendre la définition, prenons un canevas M définissant une relation d'ordre et calculons chaque étape de $suiv_M$.

$$M = \begin{vmatrix} 0 & 5 & 3 \\ 6 & 1 & 7 \\ 4 & 8 & 2 \end{vmatrix}$$

Il faut comprendre qu'une telle relation d'ordre est très erratique et n'est utilisée ici qu'à titre d'exemple. Si on voulait évaluer $suiv_M(2)$*, on ferait les calculs suivants :*

$can_M(2) = M[lig(2)][col(2)] = M[2][0] = 3$. *Comme* $can_M(x) < n^2 - 1$, *on est dans le premier cas et* $suiv_M(2) = can_M^{-1}(can_M(x) + 1) = can_M^{-1}(4) = 6$.

Pour alléger l'écriture, lorsqu'il n'y a pas de confusion possible quant au canevas M utilisé, nous notons *suiv* et *can* les fonctions $suiv_M$ et can_M. Si aucun canevas n'est défini, on utilise par défaut le canevas standard. Notons qu'en tel cas $suiv_n(x) = suiv_M(x)$ pour tout x, M le canevas standard de taille $n \times n$.

Le premier canevas que nous présentons ordonne les cases des boucles en ordre de lignes et de colonnes. L'intuition est de commencer à remplir C par la première ligne, puis la première colonne. Ensuite, les cases vides de la deuxième ligne et celles de la deuxième colonne, et ainsi de suite.

Cependant, les calculs des cycles minimaux sont faits aux positions P où $col(P) = 0$. À ces positions, $\theta(C[lig(P)-1])$ est calculé. Ainsi les cases de la ligne $lig(P)-1$ doivent être remplies lorsque l'algorithme arrive à P. Comme tous les carrés construits par nos algorithmes sont réduits, ils ont tous la même première colonne. Nous ordonnons donc ces cases pour que le nouveau canevas M_1 calcule correctement les cycles. L'exemple pour $n = 6$ clarifie très bien cela.

Exemple

Pour $n = 6$, on obtient le canevas M_1 suivant :

$$M_1 = \begin{vmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 16 & 12 & 17 & 18 & 19 & 20 \\ 24 & 13 & 21 & 25 & 26 & 27 \\ 30 & 14 & 22 & 28 & 31 & 32 \\ 34 & 15 & 23 & 29 & 33 & 35 \end{vmatrix}$$

La fonction θ est calculée aux cases où $M[P] \in \{16, 24, 30, 34, 36\}$ et à chacune d'elles, la ligne $lig(P) - 1$ est remplie en entier. Rappelons que selon la définition de suiv, $suiv(35) = 36$ et que $suiv(36) = 36$.

L'arbre construit avec le canevas M_1 présente moins de feuilles que celui construit avec M, le canevas standard (voir section 4.3.1), comme le montre la dernière ligne des tableaux 12 à 14. Cependant, le temps d'exécution de notre solution implantée est pratiquement le même qu'avec le canevas initial. Quelques essais simples nous ont permis de déterminer que les calculs permettant d'identifier les rectangles non minimaux (la fonction *rep*2) prennent plus de temps que la fouille des sous-arbres correspondants.

En effet, même si l'arbre d'exécution construit avec M_1 a moins de feuilles, il comprend beaucoup plus de noeuds aux profondeurs intermédiaires. Nous avons donc conçu un algorithme qui fusionne les avantages des canevas M et M_1. Ce nouveau canevas, que nous nommons M_2, commence par remplir la première ligne et la première colonne comme M_1. Ensuite, les cases restantes sont remplies en ligne comme pour M. Ainsi, un arbre ayant moins de feuilles est construit grâce à l'héritage de M_1. En plus, les coupures des rectangles non minimaux et des cycles minimaux permettent de

construire un arbre ayant moins de noeuds.

Exemple

Pour $n = 6$, on obtient le canevas M_2 suivant :

$$M_2 = \begin{vmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 16 & 12 & 17 & 18 & 19 & 20 \\ 21 & 13 & 22 & 23 & 24 & 25 \\ 26 & 14 & 27 & 28 & 29 & 30 \\ 31 & 15 & 32 & 33 & 34 & 35 \end{vmatrix}$$

Résultat

Les tableaux 12 à 14 présentent le nombre de feuilles des arbres d'exécution pour $5 \leq n \leq 7$ construit par l'algorithme *CompteRec* modifié. La première colonne indique la profondeur dans l'arbre d'exécution. Les trois autres colonnes indiquent le nombre de noeuds en utilisant les canevas M, M_1 et M_2.

Le lecteur remarquera que le nombre de rectangles de deux lignes est plus grand pour les canevas M_1 et M_2 que M. Cela vient du fait que la deuxième colonne est remplie avant la première case de la troisième ligne dans ces deux cas.

5.1.4 Conclusion

Nous avons essayé quelques autres canevas sans grand succès quant au nombre de noeuds. Certaines idées nous semblent cependant de bonnes veines à explorer pour construire un arbre d'exécution différent et ainsi couper d'autres branches de l'arbre. Cependant, puisqu'il semble que nous construisons déjà un arbre ayant un nombre de

$Profondeur$	$CompteRec\ (M)$	$CompteRec\ (M_1)$	$CompteRec\ (M_2)$
1	1	1	1
2	11	31	31
3	12	13	10
4	11	11	11
5	7	7	7

TAB. 12: Nombre de rectangles latins ($n = 5$) selon le canevas

$Profondeur$	$CompteRec\ (M)$	$CompteRec\ (M_1)$	$CompteRec\ (M_2)$
1	1	1	1
2	53	565	565
3	99	324	165
4	307	467	330
5	391	262	382
6	337	217	235

TAB. 13: Nombre de rectangles latins ($n = 6$) selon le canevas

feuilles et une largeur dans le même ordre que le nombre de classes d'isomorphisme, nous croyons que les gains ne feraient qu'améliorer la constante.

On pourrait, par exemple, considérer une relation d'ordre liée aux cycles. Celle-ci pourrait ordonner les carrés latins selon la taille et les éléments des cycles des carrés latins. Il serait en tel cas possible de ne construire que des lignes respectant les cycles (voir section 4.4.1) et ainsi éviter les calculs de la fonction θ de la section 3.2.3. Une telle relation d'ordre pourrait permettre de changer radicalement l'approche pour la construction de l'arbre d'exécution.

$Profondeur$	$CompteRec\ (M)$	$CompteRec\ (M_1)$	$CompteRec\ (M_2)$
1	1	1	1
2	309	15 981	15981
3	808	20 664	3490
4	10 509	85 652	22573
5	70 791	79 272	56916
6	90 548	47 547	49768
7	78 646	38 850	40358

TAB. 14: Nombre de rectangles latins ($n = 7$) selon le canevas

On peut aussi penser à une relation d'ordre qui change en fonction de la deuxième ligne. Cela pourrait permettre de définir une relation d'ordre qui serait moins longue à calculer. On pourrait ainsi trouver des représentants selon un critère différent pour chaque sous-arbre associé à un rectangle latin minimal, le plus grand par exemple.

Ce ne sont là, bien sûr, que des idées embryonnaires. Nous avons axé notre recherche sur la construction d'un arbre ayant un nombre de noeuds raisonnable plutôt que sur la structure de cet arbre.

5.2 Analyse théorique des deuxièmes lignes minimales

On remarque que le nombre de permutations possibles pour un ordre n est lié aux partages d'entiers. Donnons la définition de ces partages d'entiers pour ensuite faire le lien avec le calcul nous intéressant.

Le chapitre 3 de [13] traite des partages d'entiers et est d'ailleurs la ressource utilisée pour les définitions et exemples qui suivent. Le lecteur trouvera également des spécificités plus théoriques dans [11].

Définition 5.9 *Un partage d'entiers de m, $m \in \mathbb{N}$, est une représentation de m comme une somme de n entiers positifs $a_1, ..., a_n$. On note $P(m)$ le nombre de partages d'entiers de m.*

Exemple

Pour $m = 6$, $P(m) = 11$ et ces 11 partages d'entiers sont :

1) 6
2) $5 + 1$
3) $4 + 2$
4) $4 + 1 + 1$
5) $3 + 3$
6) $3 + 2 + 1$
7) $3 + 1 + 1 + 1$
8) $2 + 2 + 2$
9) $2 + 2 + 1 + 1$
10) $2 + 1 + 1 + 1 + 1$
11) $1 + 1 + 1 + 1 + 1 + 1$

Nous ne connaissons pas de formule pour calculer exactement $P(m)$, mais nous savons que $P(m)$ est dans $O(e^{\pi}\sqrt{\frac{2m}{3}}/m)$, selon [13]. Il existe une théorie sur les partages d'entiers ordonnés, non ordonnés, etc. Nous nous limitons ici aux partages d'entiers n'ayant qu'une restriction : pas d'éléments valant 1. Des partages précédents, nous ne gardons que :

$$6$$
$$4 + 2$$
$$3 + 3$$
$$2 + 2 + 2$$

Il existe une correspondance entre ces quatre partages de 6 et les deuxièmes lignes minimales de représentants d'ordre 6. En effet, ces quatre partages donnent exactement la taille des cycles des deuxièmes lignes minimales pour $n = 6$.

Il faut se rappeler que, sur une deuxième ligne minimale, le premier cycle contient toujours l'élément neutre et que les autres sont en ordre de taille (voir définition 3.7 page 50). Donc, le nombre de deuxièmes lignes est fortement lié au partage d'entiers.

Définition 5.10 *Soit m un entier, notons $NB(m)$ le nombre de partages de n sans l'élément neutre.*

Théorème 5.1 *Le nombre de deuxièmes lignes minimales pour une taille n est :*

$$\sum_{2<i<n} NB(n-i)$$

Preuve Ce calcul est le résultat direct des définitions. Prenons un premier cycle de taille p. Alors il y a $NB(n-p)$ façons de partager $n-p$ sans 1. C'est également le nombre de façon de diviser en cycles, les points fixes étant impossibles dans un carré latin. Il suffit d'additionner pour chaque p possible. Notons que le cas où $p = n-1$ implique $NB(n-p) = NB(n-n+1) = 0$. □

Il est donc possible de calculer une borne supérieure au nombre de deuxièmes lignes minimales. Cependant, dans le cadre du problème de génération complète auquel nous nous sommes attaqué, il est plus simple de calculer le nombre de rectangles latins minimaux de deux lignes que d'appliquer la théorie des partages d'entiers.

5.3 Analyse empirique

Nous allons dans cette section présenter l'importance de l'accélération liée aux permutations réduites pour le calcul de la fonction *rep*2. Il est difficile de quantifier celle-ci théoriquement. Dans une solution implantée, elle a permis de réduire de 75 pour cent le temps d'exécution pour $n = 7$ et $n = 8$.

Ensuite, nous présentons des mesures sur le déroulement d'une exécution. Cela permet d'appuyer notre intuition quant au fait que l'arbre d'exécution construit par notre algorithme accéléré est dépendant du nombre de classes d'isomorphisme et non du nombre de boucles.

5.3.1 La réduction des permutations pour le calcul de *rep*2

Nous avons, à la section 4.5, réduit le nombre de permutations utilisées pour le calcul de la fonction *rep*2. Nous allons comparer dans cette section le nombre de ces permutations avec les $(n-1)!$ permutations initiales.

Rappelons que la section 4.5 comprend en détail la nature des permutations préservant la deuxième ligne. Rappelons rapidement que tous les rectangles latins générés par l'algorithme *CompteRec* ont la même deuxième ligne que leur représentant. Les permutations qui permettent de préserver la deuxième ligne sont celles qui ne permutent que les éléments appartenant à des cycles de même taille. De plus, elles ne permutent le 1 qu'avec les éléments a ayant des cycles de même taille que 1 (i.e. $\theta(1) = \theta(a)$). Enfin, lorsqu'un élément d'un cycle est renommé, les autres n'ont qu'une façon de l'être.

Le calcul de *rep*2, quelles que soient les permutations utilisées (réduites ou non), ne prend pas toujours le même nombre d'itérations puisqu'il s'agit d'un algorithme de recherche. Il est donc difficile de comparer les temps d'exécution de celui-ci selon les permutations utilisées. Nous savons toutefois, par expérimentation, que l'accélération par les permutations réduites est importante. Particulièrement, si l'algorithme de génération des carrés latins utilisé implémente l'accélération par les rectangles latins minimaux de la section 4.4.2. L'implémentation que nous avons faite des permutations réduites a permis de diminuer au quart le temps d'exécution pour $n = 7$ et 8.

Nous présentons les mesures sur les permutations utilisées par *rep*2 pour les ordres 6 à 8 dans les tableaux 15 à 17. Nous commençons par l'explication des mesures de ces tableaux puis nous les présentons aux pages 115 et 116. Finalement nous les commentons en guise de conclusion.

Premièrement, fournissons l'explication des tableaux. Le nombre de permutations réduites est dépendant de la deuxième ligne. La première colonne identifie

ces deuxièmes lignes par une représentation en cycle. La deuxième colonne donne le nombre de permutations réduites fixant 1 (permutation π_3 de la définition 4.20) pour chaque deuxième ligne minimale.

Les colonnes *Représentants* et *Rejetés* se rapportent aux feuilles de l'arbre d'exécution. La colonne *Représentants* indique le nombre de feuilles de l'arbre d'exécution correspondant à des représentants. La colonne *Rejetés* dénombre les autres feuilles, toujours selon les deuxième lignes minimales.

Pour comprendre la dernière colonne, il faut se rappeler la définition 4.20 qui divise les permutations réduites en trois permutations. À chaque noeuds i, c_i permutations π_1 sont utilisées, $1 \leq c_i \leq n-1$. En effet, la permutation identité peut toujours être utilisée et en pire cas toutes les autres, élément neutre exclu, auraient les même cycles. La colonne *Candidat* est la somme de ces c_i pour la fouille du sous-arbre de chaque deuxième ligne minimale.

Exemple

Prenons la $5^{ème}$ *ligne du tableau 17.*

$2^{ème}$ *ligne*	*Permutations*	*Représentants*	*Rejetés*	*Condidats*
(120)(43)(675)	$2(1!) \times 3(1!) = 6$	16 562 664	5 049 360	28 792 392

On interprète cette ligne de la façon suivante :

- *Le sous-arbre ayant comme deuxième ligne* (120)(43)(675) *compte* 21 612 024 *feuilles (*16 562 664 + 5 049 360*).*
- 16 562 664 *de ces feuilles correspondent à des représentants.*
- *Il y a* 6 *permutations réduites ne renommant pas le* 1. *Le calcul est le suivant : On a* $\theta((120)(43)(675)) = 37788888$. *On développe ensuite* $\prod_{i=1}^{n-1} i^{(n-a_i)} \cdot (n-a_i)!$:
  $= 2^{(8-7)}1! \times 3^{(8-7)}1! \times 4^{(8-8)}0! \times 5^{(8-8)}0! \times 6^{(8-8)}0! \times 7^{(8-8)}0! \times 8^{(8-8)}0!$

$= 2(1!) \times 3(1!) = 6$ *permutations réduites fixant* 1.

- 28 792 392 *permutations* π_1 *ont été utilisées pour déterminer les* 5 049 360 *feuilles ne correspondant pas à des représentants.*

Avant de présenter les tableaux, notons que des relations d'ordre différentes (autre canevas) ou des différences d'implémentations pourraient mener à des résultats légèrement différents de ceux présentés dans les tableaux 15 à 17 qui suivent.

Ligne	*Permutations*	*Représentants*	*Rejetés*	*Condidats*
(10)(32)(54)	$2^2(2!) = 8$	38	74	138
(10)(3452)	$4(1!) = 4$	39	43	117
(120)(453)	$3(1!) = 3$	23	17	73
(1230)(54)	$2(1!) = 2$	9	6	37
(123450)	$1(1!) = 1$	0	0	0

TAB. 15: Nombre de permutations ($n = 6$)

Les mesures des tableaux 15 à 17 révèlent plusieurs choses intéressantes sur le calcul de *rep*2. Nous allons les démontrer en comparant les valeurs trouvées avec les maximums théoriques pour en apprécier l'importance.

Commençons par les résultats de la deuxième colonne qui dénombrent les permutations π_3 en fonction de la deuxième ligne. Ce sont les permutations réduites fixant le 1. On peut voir qu'elles sont moins nombreuses que l'ensemble des permutations fixant le 1 qui sont au nombre de $(n-2)!$. Même pour les deuxièmes lignes ayant le plus de permutations π_3, il s'agit d'une accélération importante.

De plus, le nombre de permutations échangeant le 1 (permutation π_1) est très faible. On sait qu'il faut essayer au moins une permutation π_1 à chaque feuille. Or on remarque, pour la plupart des sous-arbres, que seulement 30 à 50 pour cent des feuilles

Ligne	*Permutations*	*Représentants*	*Rejetés*	*Condidats*
(10)(32)(564)	$2(1!) \times 3(1!) = 6$	7682	2974	13 950
(10)(34562)	$5(1!) = 5$	5924	4256	13 369
(120)(43)(65)	$2^2(2!) = 8$	3006	2342	6 602
(120)(4563)	$4(1!) = 4$	4007	2192	8 944
(1230)(564)	$3(1!) = 3$	2177	1362	6 046
(12340)(65)	$2(1!) = 2$	821	1041	4 124
(1234560)	$1(1!) = 1$	129	607	2 447

TAB. 16: Nombre de permutations ($n = 7$)

$2^{ème}$ *ligne*	*Permutations*	*Représentants*	*Rejetés*	*Condidats*
(10)(32)(54)(76)	$2^3(3!) = 48$	5 182 735	9 796 849	15 397 567
(10)(32)(5674)	$2(1!) \times 4(1!) = 8$	25 957 543	10 387 595	44 941 445
(10)(342)(675)	$3^2(2!) = 18$	9 390 763	7 327 356	17 905 593
(10)(345672)	$6(1!) = 6$	21 598 976	11 120 343	42 300 887
(120)(43)(675)	$2(1!) \times 3(1!) = 6$	16 562 664	5 049 360	28 792 392
(120)(45673)	$5(1!) = 5$	12 801 788	4 178 436	24 099 045
(1230)(54)(76)	$2^2(2!) = 8$	4 778 355	1 810 951	8 707 735
(1230)(5674)	$4(1!) = 4$	5 895 581	2 856 985	14 189 798
(12340)(675)	$3(1!) = 3$	3 087 059	2 297 926	9 941 196
(123450)(76)	$2(1!) = 2$	906 793	1 431 411	5 475 793
(12345670)	$1(1!) = 1$	66 592	399 552	1 606 371

TAB. 17: Nombre de permutations ($n = 8$)

fouillées ont nécessité plus d'un candidat. L'accélération en temps réel du calcul de *rep*2 nous a permis de générer les carrés latins d'ordre 8 en 6 heures sur un Pentium cadencé à $900Mz$ et celles d'ordre 7 en 5 secondes.

5.3.2 Temps d'exécution

Nous parlons de *débit* au cours des prochaines pages pour signifier le nombre de représentants trouvés en fonction de temps. Nous parlons de façon analogue d'*accélération* pour les variations de ce débit.

Nous allons commencer par représenter le débit pour une fouille complète des arbres d'exécution de $n = 7$ et $n = 8$. En se référant aux deux graphiques des figures 6 et 7, notons que les courbes sont presque linéaires jusqu'à la toute fin. Cette observation vient appuyer ce que nous avons avancé à la section 4.3.5, à savoir que le temps d'exécution de l'algorithme de génération *CompteRep*2 est fonction du nombre de classes d'isomorphisme et non du nombre total de boucles.

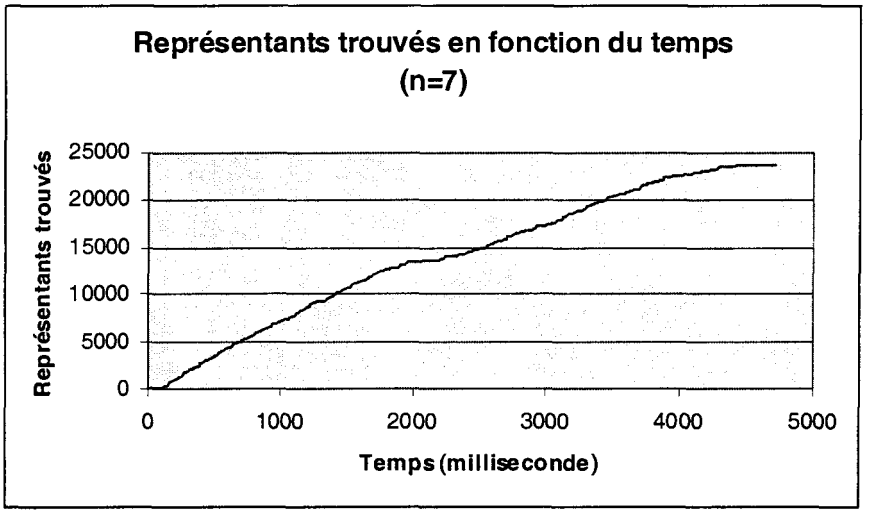

FIG. 6: Représentants trouvés en fonction du temps ($n = 7$)

Remarquons sur le graphique de la figure 7 qu'il y a des changements subtils de

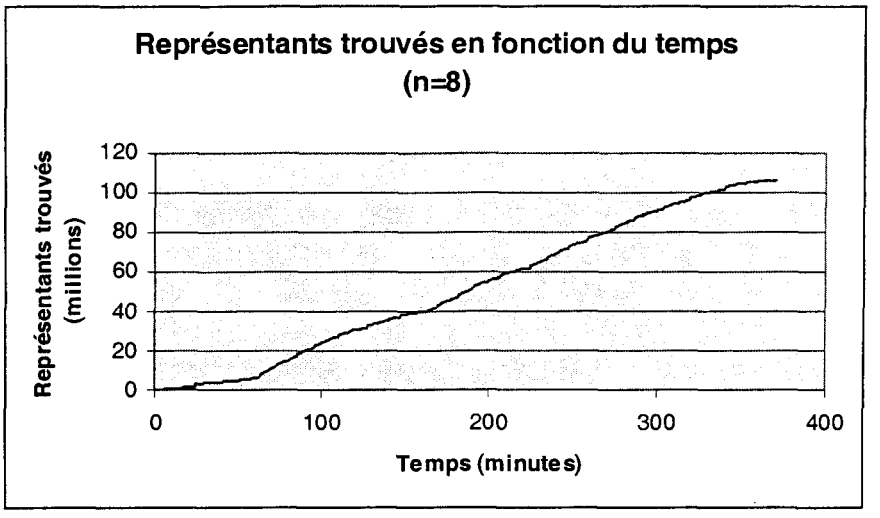

FIG. 7: Représentants trouvés en fonction du temps ($n = 8$)

pentes. Ces différents temps représentent les fouilles des différents sous-arbres ayant pour racine les rectangles latins minimaux de deux lignes. Le tableau 18 fournit les bornes en fonction du temps de l'exécution représentée à la figure 7.

On sait, comme la section 5.3.1 le démontre, que le nombre de permutations utilisées pour calculer la fonction *rep*2 sur un rectangle latin est fonction de la deuxième ligne de ce carré latin. On peut constater que le débit n'est pas fonction de ce nombre. En effet, il ne semble pas y avoir de corrélation entre la colonne *Permutations* et la colonne *Débit*.

Les graphiques des figures 6 et 7 nous portent à croire que le comportement général d'une courbe pour $n = 9$ serait similaire. Nous n'en avons pas la certitude puisqu'il semblerait que le débit soit non linéaire à la fin. En effet, à travers toutes les fouilles de l'arbre d'exécution que nous avons faites, utilisant alternativement les différentes accélérations, nous avons remarqué que le débit réduisait toujours en fin d'exécution.

Notre étude nous a permis de déterminer que ce segment, dont le débit nous semble non linéaire, correspond à la fouille du sous-arbre ayant pour racine le rectangle dont

$2^{ème}$ *ligne*	*Permutations*	*Représentants*	*Début*	*Fin*	*Débit*
(10)(32)(54)(76)	48	5 182 735	0	53	97 787.5
(10)(32)(5674)	8	25 957 543	53	124	365 599.2
(10)(342)(675)	18	9 390 763	124	164	234 769.1
(10)(345672)	6	21 598 976	164	223	366 084.3
(120)(43)(675)	6	16 562 664	223	265	394 349.1
(120)(45673)	5	12 801 788	265	301	355 605.2
(1230)(54)(76)	8	4 778 355	301	317	298 647.2
(1230)(5674)	4	5 895 581	317	339	267 981.0
(12340)(675)	3	3 087 059	339	354	205 803.9
(123450)(76)	2	906 793	354	365	82 435.7
(12345670)	1	66 592	365	370	13 318.4

TAB. 18: Statistiques de la fouille des sous-arbres (rectangles minimaux $n = 8$)

la deuxième ligne n'a qu'un cycle de taille n. Nous nommons *dernier sous-arbre* le sous-arbre du noeud correspondant à ce rectangle dans l'arbre d'exécution réduit.

Exemple

Pour $n = 9$, ce rectangle est le suivant :

$$R = \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 0 \\ \hline \end{array}$$

Nous avons effectué la fouille de ce sous-arbre afin de nous assurer que le ralentissement du débit en fin d'exécution n'est pas trop important pour $n = 9$ et qu'une fouille complète de l'arbre d'exécution pour $n = 9$ est envisageable. Le programme a mis

une centaine de jours à répondre, sur une machine cadencée à 450 MHz. Nous avons ainsi compté 777 308 122 classes d'isomorphisme et à notre connaissance il s'agit là d'un résultat original. Rappelons que le nombre de classes d'isomorphisme de boucles d'ordre 9 est 9 365 022 303 540 selon [20].

De plus, il semble que nos doutes étaient fondés puisque lors de la fouille de ce dernier sous-arbre le débit a réduit considérablement. De 3000 représentants par seconde au départ, il a réduit en fin d'exécution jusqu'à prendre plusieurs heures par représentant. Les graphiques 8 à 10 montrent le débit en fonction du temps pour n allant de 7 à 9.

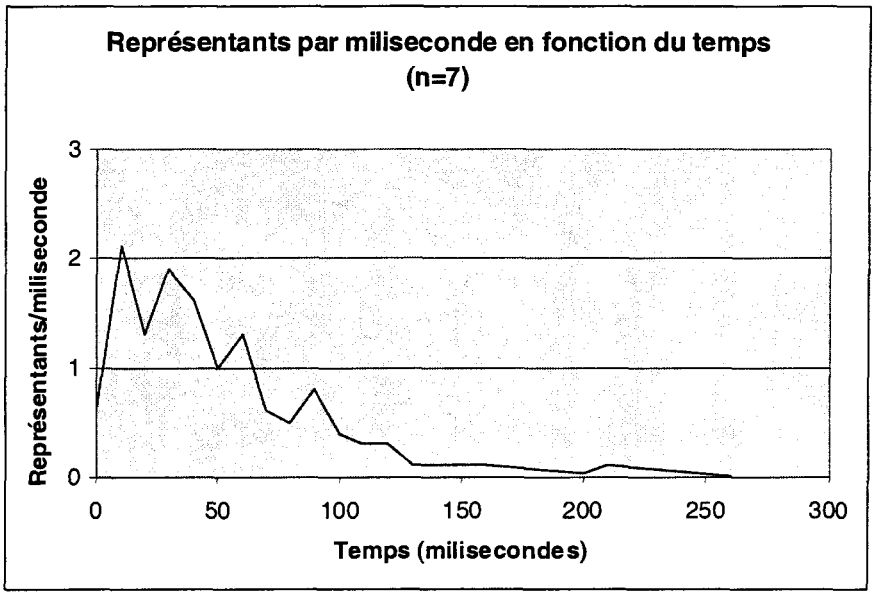

FIG. 8: Débit en fonction du temps ($n = 9$)

Expliquons que l'exécution pour $n = 7$ ne prend que 4 secondes et qu'il est ainsi difficile d'effectuer une cueillette précise de données. Quant au graphique pour $n = 9$, nous nous attendions à une exécution d'une dizaine de jours. Ce n'est qu'en nous rendant compte de notre erreur, après un mois, que nous avons commencé la cueillette de données. Il manque ainsi les mesures pour les 35 premiers jours. Ces graphiques

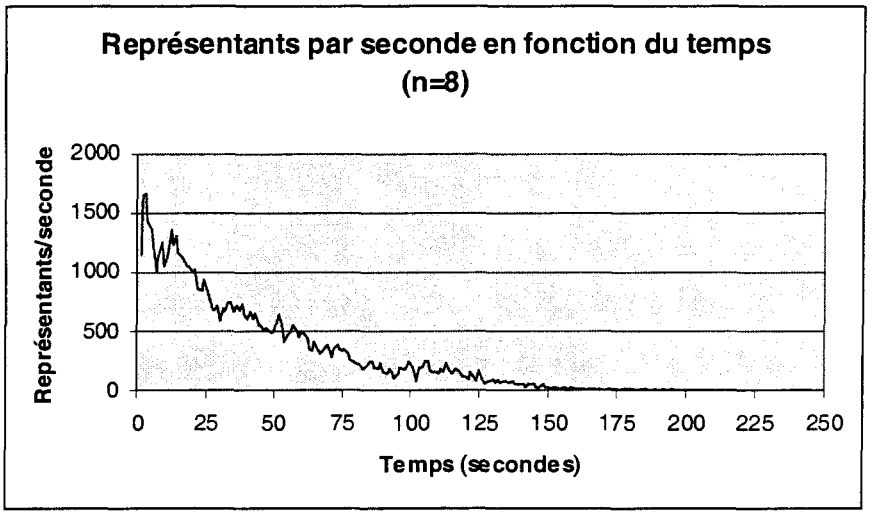

FIG. 9: Débit en fonction du temps ($n = 8$)

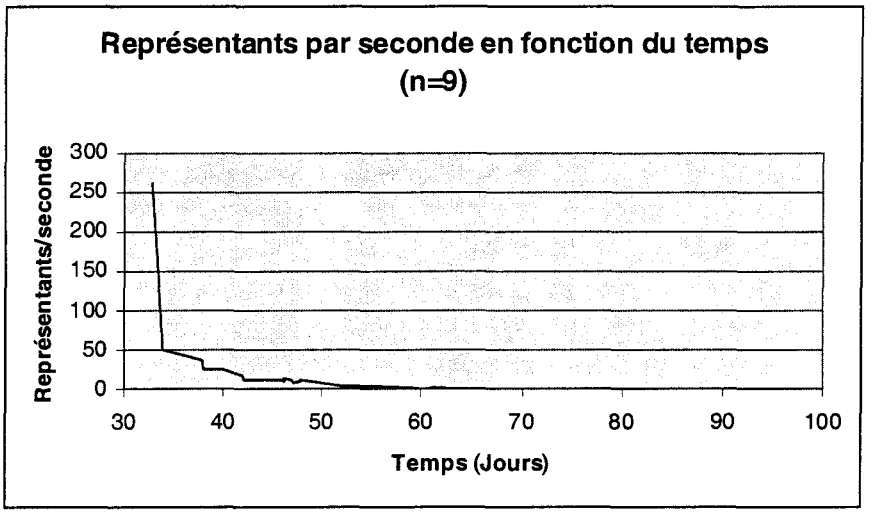

FIG. 10: Débit en fonction du temps ($n = 9$)

permettent tout de même de voir que les débits pour ces fouilles ne sont pas constants et ralentissent radicalement en fin d'exécution, particulièrement pour $n = 9$.

5.4 Conclusion

Nous croyons, à partir de l'observation des résultats de ce chapitre, pouvoir estimer le temps que prendrait une fouille complète de l'arbre d'exécution pour $n = 9$. Nous devons d'abord supposer que le comportement du débit pour $n = 9$ est similaire à celui de $n = 7$ et $n = 8$.

On peut raisonnablement le penser puisqu'il semble que ce soit vrai pour la fouille du dernier sous-arbre. De plus, des fouilles partielles des sous-arbres correspondant aux autres rectangles latins minimaux nous portent à croire que le débit est linéaire pour chacun d'eux. Nous savons que le temps pour la fouille du dernier sous-arbre ($n = 9$) est d'une centaine de jours.

Nous avons exécuté l'algorithme pendant deux jours et avons ainsi trouvé un débit de 1200 représentants par seconde pour le premier sous-arbre. Nous savons qu'au début de la fouille du dernier sous-arbre le débit était de 3000 représentants par seconde. Nous croyons qu'il est raisonnable de penser que le débit moyen se situe entre 1200 et 3000 représentants par seconde. Si on pose une moyenne de 2000 représentants par seconde, une exécution complète pour $n = 9$ prendrait environ 150 ans. En parallélisant sur quelques centaines de processeurs, il serait donc possible de ramener l'exécution à quelques mois.

Pour la taille du fichier contenant cette information, par contre, un gros problème se pose. En effet, prenons une bijection des boucles vers les entiers. Il faudrait tout de même 44 bits au minimum pour écrire chaque élément de l'image. Ce qui impliquerait un fichier d'au moins 50 000 Go pour tous les représentants de classes d'isomorphisme. Avec les technologies actuelles, c'est 1000 fois trop volumineux.

CONCLUSION

Nous présentons une conclusion en trois étapes. D'abord, nous reprenons l'ensemble des points importants de notre recherche et nous faisons ensuite ressortir les résultats qui nous semblent les plus importants. Ensuite, nous expliquons quelles utilisations pourraient être faites de notre algorithme dans le futur. Finalement, nous présentons les voies de recherche pour accélérer notre algorithme.

6.1 Discussion

L'objet de notre recherche, au départ, était de générer les classes d'isomorphisme d'ordre 8. Il existait alors un algorithme de base qui permettait de le faire. Cependant, le temps d'exécution de cet algorithme ne permettait pas, avec les technologies actuelles, de penser avoir une réponse dans un laps de temps raisonnable.

Nous avons effectué en premier lieu une étude empirique de l'exécution de l'algorithme de base. Rappelons que l'algorithme de base est récursif et effectue une fouille en profondeur d'un arbre. À certains moments, notre solution implantée fouillait pendant des temps considérables sans trouver aucun représentant pour ensuite en trouver de longues séries. Nous avons donc pensé que l'arbre d'exécution avait des branches contenant plusieurs représentants alors que d'autres n'en contenaient aucun.

C'est en analysant les représentants d'ordre 5 à 7 que nous avons découvert quelles branches contenaient des représentants et quelles branches n'en contenaient pas. Nous avons alors généralisé nos observations sous la forme de nouveaux algorithmes ne fouillant que les branches identifiées. En analysant les conditions posées pour continuer la recherche, nous avons ensuite découvert certaines propriétés communes à toutes les feuilles du nouvel arbre d'exécution. À partir de ces propriétés, nous avons finalement réduit les calculs faits à chaque feuille.

Ce nouvel algorithme était beaucoup plus rapide que l'algorithme de base mais toujours trop lent. Nous avons donc effectué une panoplie d'essais afin d'améliorer

le temps d'exécution. Plusieurs de ces essais sont restés sans succès, mais d'autres nous ont permis de réduire le temps d'exécution. La meilleure exécution pour $n = 8$ a demandé 6 heures pour s'exécuter sur une machine cadencée à 900 MHz .

Les résultats les plus importants, selon nous, sont d'abord l'atteinte de l'objectif initial soit de générer les classes d'isomorphisme d'ordre 8. Nous avons également caractérisé la deuxième ligne de tout représentant de classe. Finalement, nous avons énuméré les classes d'isomorphisme de boucles d'ordre 9 n'ayant que des cycles de taille 9.

6.2 Utilisation de l'algorithme

Nous avons effleuré l'étude des boucles d'ordre supérieur à 8 dans la conclusion du chapitre 5. Nous allons ici approfondir l'utilisation de l'algorithme pour la recherche de contre-exemples. De plus, nous discutons brièvement des algorithmes des chapitres précédents pour les paralléliser.

Enfin, il y a la génération des classes d'isotopie. Nous avons expliqué à la section 2.2.2 les raisons qui nous ont poussé à nous intéresser aux classes d'isomorphisme. Cela n'exclut pas le fait que nos algorithmes pourraient peut-être être ajustés pour générer les classes d'isotopie.

6.2.1 Étude des boucles d'ordres supérieurs à 8

Nous avons généré toutes les boucles d'ordre 8 et moins. Nous l'avons fait pour fournir le matériel permettant d'étudier et éventuellement comprendre les boucles. Cet objectif est atteint et l'algorithme que nous avons présenté permettra d'effectuer la recherche future sur les boucles.

En effet, si on pense avoir trouvé une propriété générale des boucles, il suffit d'un seul contre-exemple pour prouver qu'elle est fausse et diriger ainsi la recherche. Notre

algorithme permet de générer en temps raisonnable toutes les classes d'isomorphisme d'ordre inférieur à 9. Mais plus encore, il permet de générer un grand nombre de classes d'isomorphisme d'ordre supérieur à 8.

L'utilisation d'algorithmes de génération des chapitres 4 et 5 pour construire des algorithmes de recherche pourrait permettre de trouver des contre-exemples à certaines présumées propriétés. Prenons un exemple précis pour comprendre l'idée.

Exemple

Supposons $P : A^{|A|^2} \rightarrow \{VRAI, FAUX\}$, *une fonction calculant une propriété* P *que l'on croit vraie pour tout carré latin. On peut chercher des contre-exemples pour* P *à l'aide de l'un des algorithmes d'énumération du chapitre 4 ou 5. Il suffit de calculer* $P(C)$ *à la ligne qui retourne* 1 *puisque qu'à cette ligne un nouveau représentant de classe d'isomorphisme a été trouvé. Il suffit alors d'arrêter la récursion si* $P(C) = FAUX$, *impliquant qu'un contre-exemple a été trouvé.*

Il est possible de modifier l'algorithme de la section 5.1.2 pour fouiller l'arbre d'exécution plus globalement. En effet, il pourrait être intéressant de tester au moins un représentant pour chaque deuxième ligne. Pour le faire, rappelons que lorsqu'un point donné est atteint ($can(P) \geq F$) dans l'algorithme *CompteRec*, l'appel d'un autre algorithme y est fait (*CompteRep*5). Ainsi, il est possible d'effectuer dans cet autre algorithme la vérification de la propriété. Il faut alors décider de quelle façon ce sous-arbre sera fouillé. On pourrait par exemple y chercher un seul carré latin et le vérifier ou encore fouiller durant un temps prédéterminé.

6.2.2 Parallélisation

Dans le cas où un grand nombre d'essais à la recherche d'un contre-exemple seraient nécessaires, l'algorithme que nous présentons est récursif et facilement parallélisable. Il suffit qu'un processeur principal génère des rectangles latins et appelle la fouille du

sous-arbre d'exécution de chacun sur un processeur différent. Comme le nombre de rectangles latins est extrêmement grand, la parallélisation peut se faire à très grande échelle.

Exemple

On pourrait paralléliser sur 12 *processeurs le génération de classes d'isomorphisme de boucles d'ordre* 8. *Chacun des processeurs esclaves recevrait un des rectangles latins minimaux de* 2 *lignes. On peut trouver ces deuxièmes lignes grâce à la construction du théorème 3.4.1. À la fin de l'exécution, les différents résultats trouvés par les* 11 *processeurs esclaves seraient mis en commun.*

Bien sûr, ce n'est pas une parallélisation optimale pour l'utilisation des ressources puisque l'un des processeurs travaillerait très longtemps alors qu'un autre ne ferait que peu d'effort. Nous savons que le nombre de feuilles n'est pas dans le même ordre de grandeur pour chaque sous-arbre ayant un rectangle latin de deux lignes comme racine (voir les tableaux 15 à 17).

Cette parallélisation pourrait utiliser un très grand nombre de processeurs puisqu'à une certaine profondeur, l'arbre d'exécution a une largeur dans le même ordre que le nombre de feuilles. Comme aucune communication n'est nécessaire entre les processeurs, le débit d'exécution d'une exécution parallèle augmenterait théoriquement de façon linéaire en fonction du nombre de processeurs.

6.2.3 Génération de classe d'isotopie

Les classes d'isotopie, que nous avons définies à la section 2.2.2, forment des structures combinatoires intéressantes à générer. Il est clair qu'il serait relativement simple, à partir des classes d'isomorphisme, de générer toutes les classes d'isotopie. Cependant, cela prendrait un temps plus long encore que de générer les classes d'isomorphisme.

Il serait plus intéressant d'étudier ces classes d'isotopie de la même façon que nous avons étudié les classes d'isomorphisme et de concevoir un nouvel algorithme. Il serait peut-être possible alors de caractériser les deuxièmes lignes des représentants de classes d'isotopie comme nous l'avons fait pour les représentants de classes d'isomorphisme.

Selon la même idée, une étude des cycles pourrait être faite. De plus, les trois permutations permettant de générer des isotopies (permutation d'éléments, de lignes et de colonnes), pourraient être étudiées en profondeur. Le même genre de caractéristiques que pour les représentants de classes d'isomorphisme serait peut-être ainsi découvert. Comme nous savons que le nombre de classes d'isotopie est plus petit que le nombre de classes d'isomorphisme (voir [18]) il serait peut-être possible de générer ou d'énumérer les classes d'isotopie pour des ordres supérieurs à 8.

Bien sûr, nous n'avons fait qu'un survol des classes d'isotopie dans le cadre de notre étude et n'avons donc aucune certitude quant à la généralisation de nos algorithmes aux classes d'isotopie. Cependant, ces structures combinatoires sont liées puisque les isomorphismes sont des isotopies.

6.3 Les accélérations de l'algorithme

L'algorithme 4.4 est le fruit d'une recherche théorique pour émonder l'arbre d'exécution de même que pour réduire les calculs faits à chaque feuille. Certaines accélérations, venues de recherches empiriques, ont été implantées par des modifications mineures. Cependant, certaines idées n'ont pas été approfondies parce que les résultats ne semblaient pas valoir l'effort ou encore parce que la recherche nécessaire était trop ambitieuse. Nous présentons ici ces idées pour orienter les recherches futures.

Nous parlons d'abord de l'algorithme de génération *CompteCarre* de la page 68. La base de celui-ci est demeurée inchangée au cours des accélérations. Nous croyons que c'est le prochain point à attaquer pour améliorer le temps d'exécution des autres

algorithmes.

Deuxièmement, nous fouillons encore des branches de l'arbre d'exécution sans y trouver de représentant. Il serait peut-être possible de couper l'arbre d'exécution encore et d'arriver à ne générer que des feuilles associées à un représentants.

6.3.1 Génération

Pour ce qui est de la génération des carrés latins proprement dite, nous construisons l'arbre d'exécution avec une méthode de base. Somme toute, nous générons les carrés latins par essais et erreurs. La figure 4 de la page 62 nous montre que même pour des carrés d'ordre 4, un grand nombre de carrés latins partiels sont inutilement générés.

Le tableau 19 suivant donne le nombre de lignes partiellement complétées lors d'une exécution de *CompteRepX*. Il s'agit du cas où *Nombre* est retourné à la dernière ligne de l'algorithme sans qu'un seul appel récursif ne soit fait. En tel cas, il a été impossible de terminer la ligne commencée. Or, il est toujours possible d'ajouter une ligne à un rectangle latin (voir théorème 2.1 à la page 17).

n	*Nombre de fouilles inutiles*
3	4
4	22
5	180
6	5 693
7	1 029 510
8	5 073 612 353

TAB. 19: Lignes incomplètes

Il serait possible de générer les carrés latins ligne par ligne au lieu de case par case

puisque les accélérations ne se calculent que lorsqu'une ligne est pleine. Ainsi, entre chaque appel de la fonction de génération récursive, il y aurait n cases de plus de remplies au lieu d'une seule. On peut penser pour ce faire à plusieurs méthodes.

La première méthode consisterait à construire le graphe de la figure 2 de la page 19 et d'y trouver les 1-facteurs par un calcul déterministe. Il a été prouvé que si on prend le 1-facteur d'un graphe construit par la définition 2.18 sur un rectangle latin, ce 1-facteur calcule une nouvelle ligne pour le rectangle latin. Il serait possible d'étudier la façon de calculer ces 1-facteurs pour arriver à trouver toutes les nouvelles lignes possibles pour un rectangle latin donné en temps raisonnable.

Une seconde méthode consisterait à avoir une structure de données contenant les $n!$ permutations possibles d'un ensemble. Cette structure devrait être mise à jour à chaque appel de la fonction de génération de telle façon que les permutations représentants des nouvelles lignes possibles pour C soient connues et accessibles en temps linéaire. Ce genre de structure pourrait prendre beaucoup de mémoire et être très complexe à entretenir. Pour des ordres supérieurs à 7 et pour des rectangles de peu de lignes, nous pensons tout de même que cela pourrait être une accélération.

Présentement, nous effectuons n^n itérations pour construire chaque ligne avec la méthode de base. Nous croyons qu'il serait possible par une recherche de mieux utiliser ces n^n itérations afin de générer plus rapidement chaque ligne.

6.3.2 Étude de la troisième ligne

Nous avons caractérisé les deuxièmes lignes des représentants de classes d'isomorphisme de carrés latins réduits. C'est l'un des résultats les plus importants de notre étude au niveau de l'amélioration de temps d'exécution. Nous avons essayé sans succès de caractériser les troisièmes lignes de la même façon.

Nous nous demandons s'il est possible de construire un arbre d'exécution dont

chaque branche a au moins un représentant. En d'autres mots, concevoir un algorithme qui ne génère que des rectangles latins ayant au moins un enfant représentant, ce qui n'est pas le cas pour le moment. Prenons exemple sur le tableau de la figure 8 de la page 84. Il y a 70 791 rectangles de taille 5×7. De ceux ci, 11 415 ne respectent pas les cycles, donc l'algorithme ne fouille les sous-arbres que de 59 376 de ces rectangles latins. On peut voir qu'il y a 78 646 feuilles à l'arbre d'exécution de *CompteRep*3. Il y a donc des rectangles latins minimaux qui n'ont pas d'enfants.

La première étape vers la conception d'un algorithme ne générant que des carrés latins minimaux serait de caractériser la troisième ligne des représentants de classe d'isomorphisme. Il serait peut-être intéressant aussi de caractériser la deuxième colonne dans le cadre d'un algorithme utilisant la relation d'ordre de la section 5.1.3.

6.3.3 Application

Les classes d'isomorphisme de boucles ou de carrés latins réduits sont utilisées surtout dans les domaines théoriques. La reconnaissance de langages étant la motivation principale dans notre cas. Toutefois, nous avons essayé de trouver des applications pratiques à nos algorithmes. Nous croyons en avoir trouvé une : l'encodage de données.

L'utilisation de carrés latins pour l'encodage de données n'est pas nouvelle, quoique peu utilisée. Le lecteur peut se référer à [15] pour plus d'informations sur le sujet. Les carrés latins permettent d'encoder des données. Si un bruit modifie le message initial, il est souvent possible de reconstruire les messages sachant que ceux-ci sont composés de carrés latins (loi d'annulation).

En utilisant des carrés latins représentants de classes d'isomorphisme, il serait possible de recouvrer de l'information avec une transmission ayant subi un très fort bruit. En effet, supposons que les carrés latins C utilisés pour l'encryption sont tels que

$rep(C) = VRAI$. Alors l'information transmise avec un grand bruit pourra être reconstruite selon un critère supplémentaire. En plus de savoir que l'information est composée de carrés latins, ces carrés latins sont représentants.

Ce genre d'encryption devrait être utilisée lorsqu'une large bande passante est disponible mais subissant un très fort bruit.

Références

[1] M. Beaudry, F. Lemieux et D. Thérien, *Finite loops recognize exactly the regular open languages*, Proc. 24th, ICALP, 110-120, 1997.

[2] M. Beaudry, F. Lemieux et D. Thérien, *Star-Free Open Languages and Aperiodic Loops*, Proceedings of the International Symposium on the Theoretical Aspect of Computer Science, Springer Verlag LNCS 2010 (2001) p.87-98.

[3] F. Bédard, F. Lemieux et P. McKenzie, *Extensions to Barrington's M-Program Model*, TCS vol 107, 31-61, 1993.

[4] S. Burris et H.P. Sankappanavar, *A Course in Universal Algebre*, Springer-Verlag, 1981.

[5] H. Caussinus et F. Lemieux, *The Complexity of Computing over Quasigroups*, Foundation of Software Thechnology and Theoretical Computer Science, LNCS 880, Springer Verlag, 36-47, 1994

[6] T. Cormen, C. Leiserson, R. Rivest, *Introduction à l'algorithmique*, Dunod, 1994.

[7] S. Eilenberg, *Automata, langages and machines, Vol. B*, Academic Press, New York, 1976.

[8] R. Greenlaw et H.J. Hoover, *Fundamentals of the Theory of computation*, Morgan Kaufmann, 1998.

[9] M. Hall, Jr. *The theory of groups*, The Macmillan Company, 1959.

[10] P.R. Halmos et H.E. Vaughn, *The marriage problem*, Amer. J. Math. 72, 214-215, 1950.

[11] G.H. Hardy, E.M. Wright, *An introduction to the theory of number*, Clarendon Press, 1968.

[12] S.C. Kleene, *Representation of events in nerve nets and in finite automata*, in Automata Studies (C.E. Shannon and J. McCarthy, eds.), page 3-42, Princeton University Press, 1956.

[13] D.L. Kreher, D.R. Stinson, *Combinatorial Algorithms Generation*, Enumeration and Search, CRC Press, 1999.

[14] K. Kuratowski et A. Mostowski, *Set Theory*, North-Holland Publishing Company, 1968.

[15] C.F. Laywine et G.L. Mullen, *Discrete Mathematics using Latin Square*, Wiley Interscience, 1998.

[16] F. Lemieux, C. Moore et D. Thérien, *Polyabelian loops and Boolean Completeness*, Commentationes Matematicae Universatis Carolinae 41,4 (2000) 671-686.

[17] G. McCarty, *Topology : An Introduction wiht Application to topological groups*, Dover Publications, Inc., 1967.

[18] B.D. McKay, *Brendan McKay's home page*, http ://cs.anu.edu.au/ bdm/.

[19] B. D. McKay, *Isomorph-free exhaustive generation*, J Algorithms, 26, 306-324, 1998.

[20] B.D. McKay, A. Meynert et W. Myrvold, *Counting Small Latin Squares*, International Workshop on Groups and Graphs, Varna, Bulgarie, sept. 2001.

[21] B.D. McKay et E. Rogoyski, *Latin Square of Order 10*, Electronic Journal of Combinatorics (Vol 2), 1995.

[22] On-Line Encyclopedia of Integer Séquence, *AT&T Labs Research*, http ://www.research.att.com/~njas/sequences/.

[23] H.O. Pflugfelder, *Historical notes on loop theory*, Comment.Math.Univ.Carolinae, p359-370 et suivante, 2000.

[24] H.O. Pflugfelder, *Quasigroups and Loops : Introduction*, Heldermann, 1990.

[25] J.-E. Pin, *Finite Semigroups and Recognizable Langages : an Introduction*, Bull Research and Development, 1993.

[26] J.E. Pin, *Variétés de langages formels*, Masson, 1984.

[27] J.E. Pin et C. Reutenauer, *A conjecture on the Hall topology for the free group*, Bull. London Math. Soc. 23, 356–362, 1991.

[28] Rouquier, Raphaël, *Le monstre gentil des mathématiciens*, La recherche, octobre 2001, p50-55.

Annexe A : Implantation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


const int Taille = 7;                    //Taille des carrés
typedef int CarreLatin[Taille][Taille];


//Préparation
void InitialiserReduit (CarreLatin Carre);
void InitialiserStatistique ();


//IO Usgager
void Ajouter (CarreLatin Carre);
void Garder (CarreLatin Carre);
void Afficher (CarreLatin Carre);
void KillFichier ();
void AfficherStatistique();


//Variable globale
int Debut;
int Fin;
int AppelComplete =0;
int AppelCompleteLigneColonne =0;
int Completer[Taille];
int Rejet[2][Taille];


#include "CalculeIso.h"      //Utilisé par PlusPetit.h, Completion.h
#include "Cycle.h"           //Utilisé par PlusPetit.h, Completion.h
#include "Permutation.h"     //Utilisé par PlusPetit.h
#include "PlusPetit.h"       //Utilisé par Completion.h
#include "Completion.h"      //Utulisé par le Main.cpp
```

```
clock_t Temps;

//Principal
int main ()
{

//Calcul préliminaire
     InitialiserStatistique ();
     KillFichier();
     CarreLatin Bid;
     InitialiserReduit (Bid);

//Calcul
     Debut = time(NULL);

     int Nombre = CompleteLigneColonne (Bid, Taille*Taille,Taille+1);
     int TempsExecution = clock()/CLK_TCK;
     Fin = time(NULL);

//Impression à l'écran des statisques
     printf ("Nombre : %i,   temps execution: %i  temps reel: %i",
              Nombre,     TempsExecution,     Fin-Debut);
     AfficherStatistique();
     return 0;
}

//*****************************************************
//****    Préparation du Carre pour completion    *****
//*****************************************************
void InitialiserReduit (CarreLatin Carre)
{
     for (int y=0; y < Taille; y++)
     {
          Carre[0][y] = y;
          Carre[y][0] = y;
          for (int x=1; x<Taille; x++)
               Carre[y][x] = 0;
     }
}


void InitialiserStatistique ()
```

```
{
     for (int x=0; x< Taille; x++)
     {
          Completer [x] =0;
          Rejet[0][x] =0;
          Rejet[1][x] =0;
     }
}


//*********************************************
//****    I/O avec L'usager                *****
//*********************************************
void Ajouter (CarreLatin Carre)
{
     static int Nombre = 0;

     char Nom[] = "Vitesse?.cpp";
     Nom[7] = (char)Taille+'0';

     static FILE* Fichier = fopen (Nom, "w+");
     //if (Nombre%1000==0)
          fprintf (Fichier, "On est a %i accepte en %i\n",
                         Nombre, (clock()));

     Nombre++;
//     Garder (Carre);
     if (Nombre%1000==0)
          printf ("On est a %i accepte en %i\n",
                  Nombre, (time(NULL)-Debut));

}


void Garder (CarreLatin Carre)
{
     char Nom[] = "IsoLigne?.cpp";
     Nom[8] = (char)Taille+'0';

     static FILE* Fichier = fopen (Nom, "w+");
     static int No =1;
```

```
     No++;
     for (int y=0; y<Taille; y++)
     {
          for (int x=0; x< Taille; x++)
               fprintf (Fichier, "%i ", Carre[y][x]);
          fprintf (Fichier, "\n");
     }
     fprintf (Fichier, "******************** %i\n",No);
}


void Afficher (CarreLatin Carre)
{
     printf ("*****************************\n");
     for (int y=0; y<Taille; y++)
     {
          for (int x=0; x<Taille; x++)
               printf ("%i ", Carre[y][x]);
          printf ("\n");
     }
}

void KillFichier ()
{
     char Nom[] = "IsoLigne?.cpp";
     Nom[8] = (char)Taille+'0';

     FILE* Fichier = fopen (Nom, "w+");
     fprintf (Fichier, "*");
     fclose (Fichier);
}


void AfficherStatistique()
{
     printf ("\nNombre d'appel de Complete: %i", AppelComplete);
     printf ("\nNombre d'appel de CompleteLigneColonne: %i",
                                        AppelCompleteLigneColonne);

     printf ("\nLigne\t      PlusPetit\t       ");
     printf ("RejeterRectangle\t    RejeterCycle   \n");
     for (int x=0; x< Taille; x++)
```

```
            printf ("%i\t\t%i\t\t%i\t\t%i\n", x+1, Completer[x],
                                        Rejet[0][x], Rejet[1][x]);
}
```

```
//Opération et comparaison sur les carrés
int Comparaison       (CarreLatin Modele, CarreLatin Nouv,
                       int NoLig, int NoCol);     //public
int ComparaisonLigne  (CarreLatin Modele, CarreLatin Nouv,
                       int LigDepart, int LigFin);
int ComparaisonLigCol (CarreLatin Modele, CarreLatin Nouv,
                       int NumeroLigne);

void Appliquer        (CarreLatin Modele, CarreLatin Nouv,
                       int Perm[Taille],int NoLig, int NoCol);
void RemplaceColonne  (CarreLatin C, int Vieux, int Nouv);
void RemplaceLigne    (CarreLatin C, int Vieux, int Nouv);

//*********************************************************
//****    Calcul des isomorphe                        *****
//*********************************************************
int Comparaison (CarreLatin Modele, CarreLatin Nouv,
                        int NoLig, int NoCol)    //NoCol sert à rien
{
     int CompLigCol = ComparaisonLigCol (Modele, Nouv, 1);
     if (CompLigCol != 0)
          return CompLigCol;

     return ComparaisonLigne (Modele, Nouv, 2, NoLig-1);
}


int ComparaisonLigne (CarreLatin Modele, CarreLatin Nouv,
                      int LigDepart, int LigFin)
     for (int y= LigDepart; y <= LigFin; y++)
          for (int x=LigDepart; x <= LigFin; x++)
          {
               if (Modele[y][x] > Nouv[y][x])
                    return 1;
               if (Modele[y][x] < Nouv[y][x])
                    return -1;
          }
     return 0;
}


int ComparaisonLigCol (CarreLatin Modele, CarreLatin Nouv,
```

```
                    int NumeroLigne)
{
     for (int x=NumeroLigne; x<Taille; x++)  //Ligne Numero de Ligne
     {
          if (Modele[NumeroLigne][x] > Nouv[NumeroLigne][x])
               return 1;
          if (Modele[NumeroLigne][x] < Nouv[NumeroLigne][x])
               return -1;
     }

     for (int y=NumeroLigne+1; y<Taille; y++)
     {
          if (Modele[y][NumeroLigne] > Nouv[y][NumeroLigne])
               return 1;
          if (Modele[y][NumeroLigne] < Nouv[y][NumeroLigne])
               return -1;
     }
     return 0;
}


void Appliquer (CarreLatin Modele, CarreLatin Nouv, int Perm[Taille],
                int NoLig, int NoCol)
{
     for (int y=0; y < NoLig; y++)
          for (int x=0; x < Taille; x++)
               Nouv[Perm[y]][Perm[x]] = Perm[Modele[y][x]];
     for (y = NoLig; y < Taille; y++)
          Nouv[y][1] =
}


void RemplaceColonne (CarreLatin C, int Vieux, int Nouv)
{
     for (int x=0; x< Taille; x++)
     {
          int Tamp = C[x][Vieux];
          C[x][Vieux] = C[x][Nouv];
          C[x][Nouv] = Tamp;
     }
}
```

```
void RemplaceLigne(CarreLatin C, int Vieux, int Nouv)
{
     for (int x=0; x< Taille; x++)
     {
          int Tamp = C[Vieux][x];
          C[Vieux][x] = C[Nouv][x];
          C[Nouv][x] = Tamp;
     }
}
```

```
//Cycle
int CompareCycle (int Ligne[Taille], int CycleEtalon[Taille]);
void CalculerToutLeVecteurCycle (int Ligne[Taille],
                                 int Cycle[Taille]);
void CalculerLesCycles          (int Ligne[Taille], bool Vue[Taille],
                                 int Cycle[Taille]);
int CalculerProchainCycle       (int Ligne[Taille], bool Vue[Taille]);
int CalculerLeCycle             (int Ligne[Taille], bool Vue[Taille],
                                 int Debut);

//*********************************************************
//****    Calcul des cycles                          *****
//*********************************************************
int CompareCycle (int Ligne[Taille], int CycleEtalon[Taille])
{
     int Cycle[Taille] = {0};
     bool Vue[Taille] = {false};
     if ((Cycle[0] = CalculerLeCycle (Ligne, Vue, 0)) -
                         CycleEtalon[0] != 0)
          return Cycle[0] - CycleEtalon[0];

     CalculerLesCycles (Ligne, Vue, Cycle);

     for (int y=2; y < Taille; y++)
     {
          if (CycleEtalon[y] < Cycle[y]) //Le premier est plus petit
               return -1;
          if (CycleEtalon[y] > Cycle[y]) //Le premier est plus grand
               return +1;
     }
     return 0;                          //Les cycles sont équivalents
}


void CalculerToutLeVecteurCycle (int Ligne[Taille],
                                 int Cycle[Taille])
{
     for (int y=0; y< Taille; y++) Cycle[y] =0;
     bool Vue[Taille] = {false};
     Cycle[0] = CalculerLeCycle (Ligne, Vue, 0);
     CalculerLesCycles (Ligne, Vue, Cycle);
```

```
}


void CalculerLesCycles (int Ligne[Taille], bool Vue[Taille],
                        int Cycle[Taille])
{
     int Long =0;
     while ( (Long = CalculerProchainCycle (Ligne, Vue)) != 0)
          Cycle[Long]++;
}


int CalculerProchainCycle (int Ligne[Taille], bool Vue[Taille])
{
     int Pos = 1;
     while (Vue[Pos] && Pos < Taille) Pos++;
     if (Pos >= Taille) return 0;
     return CalculerLeCycle (Ligne, Vue, Pos);
}


int CalculerLeCycle (int Ligne[Taille], bool Vue[Taille], int Debut)
{
     int Nombre=1;
     int Pos=Debut;
     while (Ligne[Pos] != Debut)
     {
          Vue[Pos] = true;
          Pos = Ligne[Pos];
          Nombre++;
     }
     Vue[Pos] = true;
     return Nombre;
}
```

```
//Permutation
bool IncrementerTotal (int Perm[Taille], int Cycle[Taille],
                       int Coupe);

bool IncrementerLesCycles (int Perm[Taille], int Cycle[Taille]);
bool IncrementerUnCycle (int Perm[Taille], int Debut, int Fin);
bool PermuterCycle      (int Perm[Taille], int Cycle[Taille],
                         int Depart);
void MettreCycleEnOrdre (int Perm[Taille], int Debut,
                         int NombreCycle, int TailleCycle);
void EchangerCycle (int Perm[Taille], int Premier, int Deuxieme,
                    int TailleCycle);
int PositionCycle (int Cycle[Taille], int Max);

//*********************************************************
//****    Permutation                                  *****
//*********************************************************
bool IncrementerTotal (int Perm[Taille], int Cycle[Taille], int Coupe)
{
     bool Reponse = false;

     if (IncrementerLesCycles (Perm, Cycle))
          Reponse = true;

     if (!Reponse)
          if ( PermuterCycle (Perm, Cycle, Taille-2) )
               Reponse = true;

     if (Reponse)
          for (int x=1; x < Coupe; x++)
               if (Perm[x] >= Coupe)
                    return IncrementerTotal(Perm, Cycle, Coupe);
     return Reponse;
}


bool IncrementerLesCycles (int Perm[Taille], int Cycle[Taille])
{
     int CycleRendu = Taille-2;
     while (Cycle[CycleRendu] == 0) CycleRendu --;
```

```
     int DebutCycle = Taille;
     while (CycleRendu > 1)
     {
          int CyclePareil = Cycle[CycleRendu];
          while (CyclePareil > 0)
          {
               DebutCycle -= CycleRendu;
               if (IncrementerUnCycle (Perm, DebutCycle,
                                       DebutCycle + CycleRendu-1))
                    return true;
               CyclePareil--;
          }
          CycleRendu--;
     }
     return false;
}


bool IncrementerUnCycle (int Perm[Taille], int Debut, int Fin)
{
     bool Retour = true;

     if (Perm[Debut] > Perm[Debut+1]) Retour = false;

     int Premier = Perm[Debut];
     for (int y=Debut; y <Fin; y++)
          Perm[y] = Perm[y+1];
     Perm[Fin] = Premier;
     return Retour;
}


bool PermuterCycle (int Perm[Taille], int Cycle[Taille],
                    int TailleCycle)
{
     if (TailleCycle <= 1) return false;

     while (Cycle[TailleCycle] <= 1) TailleCycle --;

     if (TailleCycle <= 1) return false;

     int PosCycle = PositionCycle (Cycle, TailleCycle);
```

```
    int CycleAChanger = PosCycle + (Cycle[TailleCycle]-2) *
                                        (TailleCycle);
    int Pass=1;
    while (Perm[CycleAChanger + TailleCycle] < Perm[CycleAChanger])
    {
        Pass ++;
        CycleAChanger -= TailleCycle;
    }

    if (CycleAChanger < PosCycle)
    {
        MettreCycleEnOrdre (Perm, PosCycle, Cycle[TailleCycle],
                            TailleCycle);
        return PermuterCycle (Perm, Cycle, TailleCycle-1);
    }

    int PlusPetit = CycleAChanger + TailleCycle;
    int y = PlusPetit;
    while (y <= PosCycle + (Cycle[TailleCycle]-1) * TailleCycle)
    {
        if (Perm[y] > Perm[CycleAChanger] &&
                Perm[y] < Perm[PlusPetit])
            PlusPetit =y;
        y++;
    }
    EchangerCycle (Perm, PlusPetit, CycleAChanger, TailleCycle);
    MettreCycleEnOrdre (Perm, CycleAChanger+TailleCycle, Pass,
                        TailleCycle);
    return true;
}


void EchangerCycle (int Perm[Taille], int Premier, int Deuxieme,
                    int TailleCycle)
{
    for (int y=Premier, x = Deuxieme;
        y < Premier+TailleCycle; y++, x++)
    {
        int Tamp = Perm[y];
        Perm[y] = Perm[x];
        Perm[x] = Tamp;
```

```
     }
}


void MettreCycleEnOrdre (int Perm[Taille], int Debut, int NombreCycle,
                         int TailleCycle)
{
     int Pos = Debut;
     while (Pos < (Debut + (NombreCycle-1)*TailleCycle ) )
     {
          if (Perm[Pos] > Perm [Pos +TailleCycle])
          {
               EchangerCycle (Perm, Pos, Pos+TailleCycle, TailleCycle);
               if (Pos > Debut) Pos = Pos - TailleCycle;
          }
          else
               Pos = Pos + TailleCycle;
     }
}


int PositionCycle (int Cycle[Taille], int Max)
{
     int Tot = Cycle[0];
     int y=2;
     while (y < Max)
     {
          Tot = Tot + Cycle[y] * y;
          y++;
     }
     return Tot;
}
```

```
//Calcul du plus petit
bool PlusPetit (CarreLatin Carre, int Cycle[Taille],
                int NoLig, int NoCol);

void TrouverPermutation (int Cycle[Taille], int LigneSource[Taille],
                         int Permu[Taille]);
void TrouverCycle       (int Ligne[Taille], int CycleSigne[Taille]);
void PlacerCycleSigneEnOrdre (int CycleSigne[Taille],
                              int NombreCycle[Taille]);
void EchangerCycleDifferent  (int CycleSigne[Taille], int Cycle1,
                              int Cycle2, int Debut);
void Echanger (int Perm[Taille], int N1, int N2);
int abs       (int N);

//*********************************************************
//****    Calcul Plus petit                          *****
//*********************************************************
bool PlusPetit (CarreLatin Carre, int Cycle[Taille],
                int NoLig, int NoCol)
{
     for (int Futur1=1; Futur1 < NoLig; Futur1++)
     {
          if (CompareCycle (Carre[Futur1], Cycle) <= 0)
          {
               int Permu[Taille];
               for (int x=0; x< Taille; x++) Permu[x] = x;

               int Cycle[Taille];
               CalculerToutLeVecteurCycle (Carre[1], Cycle);

               TrouverPermutation (Cycle, Carre[Futur1], Permu);
               CarreLatin Modele;
               Appliquer (Carre, Modele, Permu, Taille, 2);

               CarreLatin Copie;
               for (x=0; x< Taille; x++) Permu[x] = x;
               do
               {
                    Appliquer (Modele, Copie, Permu, NoLig, 2);
                    if (Comparaison (Carre, Copie, NoLig, 00) > 0)
                         return false;
               }
```

```
                while (IncrementerTotal(Permu, Cycle, NoLig));
            }
        }
        return true;
}


void TrouverPermutation (int Cycle[Taille], int LigneSource[Taille],
                         int Permu[Taille])
{
        int CycleSigne[Taille];
        TrouverCycle (LigneSource, CycleSigne);

        for (int y=0; y < Taille; y++)
            Permu[abs (CycleSigne[y])] = y;
}


void TrouverCycle (int Ligne[Taille], int CycleSigne[Taille])
{
        bool Vu[Taille] = {false};
        int OrdreCycle[Taille/2] = {0};
        int OrdreRendu =0;
        int NextCycle =1;
        int x=Ligne[0];
        int Compte=1;
        int TailleCycle=1;

        CycleSigne[0] = 0;Vu[0]=true;
        while (Compte < Taille)
        {
            if (!Vu[x])
            {
                Vu[x] = true;
                CycleSigne[Compte] = x * NextCycle;
                Compte++; TailleCycle++;
                x = Ligne[x];
            }
            else
            {
                NextCycle = -NextCycle;
                while (Vu[x]) x++;
```

```
                OrdreCycle[OrdreRendu++] = TailleCycle;
                TailleCycle=0;
           }
      }
      OrdreCycle[OrdreRendu++] = TailleCycle;
      TailleCycle=1;
      PlacerCycleSigneEnOrdre (CycleSigne, OrdreCycle);
}


void PlacerCycleSigneEnOrdre (int CycleSigne[Taille],
                              int OrdreCycle[Taille])
{
      int OrdreRendu=1;
      int Total = OrdreCycle[0];
      while (Total < Taille && OrdreCycle[OrdreRendu+1] != 0)
      {
           if (OrdreCycle[OrdreRendu] > OrdreCycle[OrdreRendu+1])
           {
                EchangerCycleDifferent (CycleSigne,
                                        OrdreCycle[OrdreRendu],
                                        OrdreCycle[OrdreRendu+1],
                                        Total);
                Echanger (OrdreCycle, OrdreRendu, OrdreRendu+1);
                if (OrdreRendu > 1 ) OrdreRendu --;
           }
           else
           {
                Total += OrdreCycle[OrdreRendu];
                OrdreRendu++;
           }
      }
}


void EchangerCycleDifferent (int CycleSigne[Taille], int Cycle1,
                             int Cycle2, int Debut)
{
      //Cycle1 > Cycle2 sinon ça marche pas
      int Tamp[(int)(Taille/2)] = {0};
      for (int y=0; y < Cycle1; y++)
           Tamp[y] = CycleSigne[Debut+y];
```

```
     for (y=0; y < Cycle2; y++)
          CycleSigne[Debut+y] = CycleSigne[Debut+Cycle1+y];

     for (y=0; y < Cycle2; y++)
          CycleSigne[Debut+Cycle1+y] = Tamp[y];
}


void Echanger (int Perm[Taille], int N1, int N2)
{
     int Tamp = Perm[N1];
     Perm[N1] = Perm[N2];
     Perm[N2] = Tamp;
}


int abs (int N)
{
     if (N >= 0)
          return N;
     else
          return -N;
}
```

```
//Ligne2 Et Colonne2

int CompleteLigneColonne (CarreLatin Carre, int Fin, int Current);
int Incrementer          (int Current);

//Calcul de completion
int Complete            (CarreLatin Carre, int Fin, int Current,
                         int NoCol, int CycleLigne2[Taille]);
void TrouverNombreLibre (CarreLatin Carre, int Cur,
                         bool NombreLibre[Taille]);

//*****************************************************
//****    Ligne 2 Colonne 2                       *****
//*****************************************************
int CompleteLigneColonne (CarreLatin Carre, int Fin, int Current)
{
     int Nombre = 0;
     AppelCompleteLigneColonne++;

     if (Current == Taille*2 +2 || Current == Taille*3 +3)
     {
          Completer[Current/Taille-1]++;

          int Cycle[Taille]={0};
          CalculerToutLeVecteurCycle(Carre[1], Cycle);

          if (!PlusPetit (Carre, Cycle, Current/Taille,
                                Current%Taille) )
          {
               Rejet[0][Current/Taille-1]++;
               return 0;
          }
          else
               return Complete (Carre, Taille*Taille,
                                        Current, 2, Cycle);
     }

     bool NombreLibre[Taille];
     TrouverNombreLibre (Carre, Current, NombreLibre);
     for (int x=0; x< Taille; x++)
     {
          Carre[(int)(Current/Taille)][Current%Taille] = x;
```

```
          if (NombreLibre[x])
               Nombre += CompleteLigneColonne
                              (Carre, Fin, Incrementer(Current));
     }
     return Nombre;
}


int Incrementer (int Current)
{
     if (Current == Taille*Taille-1) return Taille * Taille;

     int PosY = (int)(Current / Taille);
     int PosX = Current % Taille;

     if ( PosX >= PosY ) PosX++;
     else PosY++;

     if (PosX == Taille)
          return Current + PosY +1;

     if (PosY == Taille)
          return (PosX+1) * Taille + PosX+1;

     return PosY * Taille + PosX;
}

//*********************************************************
//****    Complete                                    *****
//*********************************************************
int Complete (CarreLatin Carre, int Fin, int Current, int NoCol,
              int CycleLigne2[Taille])
{
     AppelComplete++;
     int Nombre=0, Colonne = Current%Taille, Ligne = Current/Taille;

     if (Colonne ==0)
     {
          Completer[Current/Taille-1]++;
          if (CompareCycle(Carre[Ligne-1], CycleLigne2) < 0)
          {
               Rejet[1][Ligne-1]++;
```

```
                return 0;
            }
            if (!PlusPetit (Carre, CycleLigne2, Ligne, Taille))
            {
                Rejet[0][Ligne-1]++;
                return 0;
            }
            Current+=NoCol;          //Saut de deux premierès colonnes
            Colonne = Current%Taille;
            Ligne = Current/Taille;
        }

        if (Current >= Fin)
        {
            if (PlusPetit (Carre, CycleLigne2, Taille, Taille))
            {
                Ajouter (Carre);
                return 1;
            }
            else
            {
                Rejet[0][Taille-1]++;
                return 0;
            }
        }

        bool NombreLibre[Taille];
        TrouverNombreLibre (Carre, Current, NombreLibre);
        for (int x=0; x < Taille; x++)
        {
            Carre[Ligne][Colonne] = x;
            if (NombreLibre[x]) Nombre += Complete (Carre, Fin, Current+1,
                                                    NoCol, CycleLigne2);
        }
        return Nombre;
}


void TrouverNombreLibre (CarreLatin Carre, int Cur,
                         bool NombreLibre[Taille])
{
        int PosX = Cur % Taille;
```

```
        int PosY = (int)(Cur /Taille);

        for (int x=0; x < Taille; x++)
             NombreLibre[x] = true;
        for (x=0; x < PosX; x++)
             NombreLibre[Carre[PosY][x]] = false;;
        for (int y=0; y< PosY; y++)
             NombreLibre[Carre[y][PosX]] = false;
}
```