Vérification formelle de programmes de génération de données structurées

Richard Genestier

▶ To cite this version:

Richard Genestier. Vérification formelle de programmes de génération de données structurées. Performance et fiabilité [cs.PF]. Université de Franche-Comté, 2016. Français. NNT : 2016BESA2041 . tel-01446896v2

HAL Id: tel-01446896
https://theses.hal.science/tel-01446896v2

Submitted on 2 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques

UNIVERSITÉ DE FRANCHE-COMTÉ

N° referencenumber

THÈSE présentée par

RICHARD GENESTIER

pour obtenir le

Grade de Docteur de l'Université de Franche-Comté

Spécialité : **Informatique**

Vérification formelle de programmes de génération de données structurées

Soutenue publiquement le 1er décembre 2016 devant le jury composé de :

OLGA KOUCHNARENKO	Directeur	Professeur à l'Université de Franche-Comté
ALAIN GIORGETTI	Encadrant	Maître de Conférences à l'Université de Franche-Comté
MARIE-LAURE POTET	Rapporteur	Professeur à Grenoble INP
PASCAL SCHRECK	Rapporteur	Professeur à l'Université de Strasbourg
LOÏC CORRENSON	Examinateur	Expert senior au CEA Tech, institut LIST
VINCENT VAJNOVSZKI	Examinateur	Professeur au LE2I, Université de Bourgogne

Remerciements

Ne le cachons pas : ces quatre années furent ô combien enrichissantes, mais ce ne fut pas facile tous les jours de mener de concert plusieurs vies ...

Les conditions de déroulement de ma thèse – à distance – font que j'ai côtoyé régulièrement peu de collègues au sein du Département d'Informatique et des Systèmes Complexes de l'Université de Franche-Comté. Je remercie néanmoins tous les membres de ce laboratoire, et tout particulièrement Jacques Julliand et Pierre-Cyrille Héam pour leur accueil et leur soutien.

Je tiens à remercier chaleureusement mes rapporteurs, Marie-Laure Potet et Pascal Schreck, pour avoir accepté de consacrer leur temps précieux à décortiquer mon manuscrit, et pour leurs remarques avisées sur mes travaux. Je remercie également vivement mes examinateurs, Vincent Vajnovszki et Loïc Correnson, qui me font l'honneur de leur présence dans le jury de cette thèse.

Mes remerciements les plus vifs vont bien évidemment à mes encadrants, qui m'ont accompagné tout au long de ces années. Je remercie Olga Kouchnarenko d'avoir accepté de diriger mes travaux de thèse. Malgré ses obligations, elle a su rester disponible et ses conseils m'ont été précieux.

Comment exprimer mes remerciements infinis à mon encadrant Alain Giorgetti ? Le temps monumental accordé à mon travail, les Skype enrichissants qui ont émaillé ces quatre années (cinq avec le Master) ? Sa gentillesse et la patience hors norme dont il a su faire preuve face à mes nombreuses incompréhensions et mes atermoiements ? Sa rigueur dans ses multiples relectures de mon improbable prose ? Je le remercie d'avoir cru en moi alors que j'avais – et que j'ai toujours – tant à apprendre. J'espère avoir su tirer les enseignements de ses innombrables conseils.

J'adresse mes plus sincères remerciements à Catherine Dubois, qui arpente toutes les conférences du domaine, pour ses précieux conseils, ses encouragements et sa collaboration enrichissante lors de notre publication commune. Je remercie également Noam Zeilberger pour son expertise lors d'une publication (malheureusement non aboutie), Guillaume Petiot pour son aide apportée avec StaDy, ainsi que Nikolaï Kosmatov pour sa bienveillance.

Je tiens à remercier également mes collèques de l'IEM à Dijon pour leur bonne humeur et leur convivialité, rendant ces années moins difficiles. Je remercie tout particulièrement Jean-Luc Baril pour notre collaboration de recherche enrichissante et fructueuse, son aide précieuse, et ses inestimables encouragements dans mes nombreuses phases de doute.

Comment ne pas remercier ma famille ? Je suis très touché que de nombreux membres de ma famille aient fait le déplacement dans la lointaine cité bisontine pour assister à ma soutenance. Merci à mes filles Alice, Lucie, Ninon et Mina pour leur joie de vivre. Elles garderont je pense de ces années l'image d'un papa très bizarre que s'asseyait

derrière son ordinateur dès qu'il avait cinq minutes de libre ... Ma compagne Isabelle a dû vaillamment "assurer le quotidien", en plus de ses prenantes obligations, lors de mes phases d'isolement dans le bureau pendant les "coups de bourre" de rédaction d'articles et du manuscrit. Pendant ces années denses, elle a toujours su rester disponible, me rassurer et m'encourager. Je l'en remercie profondément.

SOMMAIRE

1 Introduction **1**

1.1 Cadre d'étude . . . 3
1.2 Problématique . . . 3
1.3 Contributions . . . 4
1.4 Plan du mémoire . . . 6

2 Contexte et état de l'art **9**

2.1 Eléments de combinatoire énumérative . . . 9
2.2 Méthodes et outils du génie logiciel . . . 22
2.3 Formalisations existantes . . . 28
2.4 Conventions et notations . . . 29

3 Génération de tableaux structurés **31**

3.1 Introduction . . . 31
3.2 Principes et exemple . . . 33
3.3 Génération par filtrage . . . 40
3.4 Bibliothèque de générateurs formellement vérifiés . . . 44
3.5 Conclusion et perspectives . . . 50

4 Opérations sur les permutations **53**

4.1 Opérations sur les permutations . . . 53
4.2 Spécification et preuve . . . 56
4.3 Propriétés de l'insertion . . . 61
4.4 Synthèse et perspectives . . . 63

5 Génération de cartes étiquetées **65**

5.1 Générateur efficace d'involutions sans point fixe . . . 66
5.2 Transitivité d'une paire de permutations . . . 72
5.3 Générateur générique de couples de tableaux . . . 76
5.4 Générateur générique de couples transitifs . . . 77

5.5 Génération exhaustive de cartes et hypercartes étiquetées 78
5.6 Carte locale 79
5.7 Résultats expérimentaux 80
5.8 Synthèse et perspectives 82

6 Codage des cartes planaires enracinées 83
6.1 Historique 83
6.2 Mots codant les cartes planaires enracinées 84
6.3 Formalisation des mots de Dyck 88
6.4 Générateur séquentiel de tableaux de hauteurs 90
6.5 Générateur de couples de tableaux de hauteurs 95
6.6 Générateur séquentiel de mélanges non croisés 99
6.7 Validation des générateurs 101
6.8 Synthèse et perspectives 102

7 Classes de cartes planaires 103
7.1 Aspect expérimental 105
7.2 Résultats préliminaires 107
7.3 Equivalence modulo $\pi \in \{a, b, \bar{a}, \bar{b}\}$ 109
7.4 Equivalence modulo $\pi \in \{a\bar{a}, b\bar{b}, \bar{a}a, \bar{b}b\}$ 111
7.5 Equivalence modulo $\pi \in \{aa, \bar{a}\bar{a}, bb, \bar{b}\bar{b}\}$ 113
7.6 Equivalence modulo $\pi \in \{ab, ba, \bar{a}\bar{b}, \bar{b}\bar{a}\}$ 115
7.7 Synthèse et perspectives 117

8 Test et preuve pour les cartes 119
8.1 Test de conjectures Coq 120
8.2 Etude de cas des cartes locales 127
8.3 Synthèse 136

9 Génération de cartes non étiquetées 139
9.1 Construction des cartes enracinées 140
9.2 Construction des cartes planaires enracinées 144
9.3 Variante pour les cartes planaires enracinées 149
9.4 Synthèse et perspectives 154

10 Hypercartes enracinées 157
10.1 Algorithme OMR 157

10.2 Implémentation en C 159
10.3 Spécification et preuves 160
10.4 Tests 162
10.5 Caractérisation des hypercartes enracinées 163
10.6 Perspectives 164

11 Conclusion **167**
11.1 Synthèse 167
11.2 Perspectives 169

1

Introduction

Les systèmes informatisés sont omniprésents de nos jours. Ils constituent la clé de voûte de nombreux secteurs, tels que le secteur bancaire, l'industrie, ou le domaine médical. En cas de dysfonctionnement de ces systèmes, des dommages importants peuvent survenir, pouvant notamment mettre des vies humaines en danger. Les causes de ces dysfonctionnements sont en général basiques. Ce sont par exemple des erreurs à l'exécution dues à la perte de précision lors de la conversion d'un entier vers un nombre flottant, à l'accès à une zone mémoire indisponible d'un tableau, ou bien tout simplement à une division par zéro. Cependant, détecter de telles erreurs est une tâche difficile. Il existe différentes méthodes pour vérifier la correction d'un programme. Beaucoup de bugs sont mis en évidence par les méthodes de test logiciel. Cependant, de telles méthodes ne peuvent pas assurer qu'un programme ne contient pas de bugs, ou bien qu'il satisfait une certaine propriété. Une autre approche de ce problème est la vérification formelle de programmes, qui fournit une preuve mathématique de leur correction.

La vérification par preuve utilise des fondements mathématiques et logiques pour prouver des propriétés de programmes [Floyd, 1967, Hoare, 1969]. Parmi ces propriétés, nous ne considérons ici que celles, dites fonctionnelles ou comportementales (*behavioral*), qui décrivent la relation entre l'entrée et la sortie d'un programme. Ces propriétés peuvent être formalisées dans un langage formel de spécification comportementale (BISL, pour *Behavioral Interface Specification Language*) [Hatcliff et al., 2012]. Dans ce contexte, on parle de "preuve de programme" pour désigner une démonstration qu'un programme correspond à sa spécification fonctionnelle.

La méthode classique de preuve de correction d'un programme impératif consiste (i) à munir ce programme d'assertions, telles que les invariants et variants de boucle, (ii) à réduire cette preuve de correction à la vérification de formules logiques (où le programme n'apparaît plus), et (iii) à démontrer la validité de ces formules. Ces étapes peuvent s'effectuer manuellement, mais dans les faits elles sont effectuées avec l'aide de la machine, et même totalement automatisées pour certaines d'entre-elles. L'étape (i) peut être assistée par l'utilisation d'outils (incomplets) de synthèse d'invariants. L'étape (ii) peut être entièrement automatisée par un générateur de conditions de vérification. L'étape (iii) peut être grandement facilitée par l'utilisation de prouveurs de théorèmes chargés de démontrer mathématiquement les conditions de vérification. Il peut s'agir de prouveurs automatiques, tels que des solveurs SMT, ou bien de prouveurs interactifs (ou assistants de preuve) où l'utilisateur doit guider la preuve. Ces outils seront détaillés dans le chapitre 2.

La spécification formelle et la preuve de programmes rendent la conception de pro-

grammes plus rationnelle, augmentent leur qualité et accroissent la confiance du développeur dans la correction de sa production. Cependant, ces techniques ont la réputation d'être difficiles à mettre en oeuvre et réservées à des experts. En raison de leur fort coût, elles ne sont pratiquées que dans des domaines à fort enjeu de sûreté ou de sécurité. Des travaux récents [Leroy, 2009, Oe et al., 2012, Carlier et al., 2012] remettent en cause ces croyances et rendent ces pratiques vertueuses accessibles à d'autres domaines du développement logiciel. L'obstacle de l'apprentissage d'un langage spécifique comme B ou Eiffel [Meyer et al., 1987] a été levé par la création de langages de spécification proches des langages de programmation les plus répandus, tels que C ou Java. L'effort d'intégration dans les environnements de développement a été initié et se poursuit. Il est particulièrement avancé dans Dafny [Leino, 2010]. L'assistance à la spécification et à la preuve, jusqu'à leur automatisation, sont améliorées par de nombreux progrès, que ce soit en synthèse d'invariants, dans les procédures de décision implémentées dans les prouveurs automatiques, ou dans les couplages avec des analyses dynamiques [Kovács et al., 2009, Reynolds et al., 2016, Petiot et al., 2014].

Pour que la preuve de programmes soit plus largement pratiquée, il est souhaitable que la correction d'un programme suffisamment spécifié soit prouvée automatiquement. Comme le problème général de la preuve de programmes est indécidable, nous souhaitons automatiser cette troisième étape pour un sous-problème, portant sur certains programmes et certains contrats. Théoriquement, il s'agit d'établir que les conditions de vérification de ces programmes avec contrats sont dans un fragment logique décidable. En pratique, on a observé que les prouveurs qui implémentent des procédures de décision pour de nombreux fragments savent aussi démontrer des formules qui n'appartiennent à aucun fragment décidable connu. Pour évaluer leurs limites pratiques, nous abordons donc la question de l'automatisation de manière empirique, en soumettant des preuves de programmes de plus en plus complexes à un outil de vérification qui utilise ces prouveurs, à travers diverses études de cas.

Une donnée structurée est une structure complexe, telle qu'un arbre, un graphe, une liste ou un tableau. Dans ce mémoire, nous nous limitons aux programmes manipulant des données structurées constituées d'un ou plusieurs tableaux. Ainsi, par différentes études de cas, ce travail de thèse tente d'évaluer la faisabilité de démonstrations automatiques de programmes manipulant des tableaux d'entiers.

Le cadre d'investigation choisi est le domaine de la combinatoire énumérative. Ce domaine n'est encore à ce jour que peu perméable aux pratiques du génie logiciel. En d'autres termes, la recherche en combinatoire énumérative est essentiellement une activité de réflexion humaine et de démonstration sur papier, demandant beaucoup d'expérience et d'intuition. Nous pensons que cette tâche délicate peut être facilitée par une formalisation machine des problèmes dès le début de leur étude, accompagnée d'une utilisation systématique d'outils logiciels et de méthodes formelles de spécification et de vérification.

Nous présentons dans la partie 1.1 le domaine d'étude abordé dans notre travail. La problématique générale est ensuite exposée dans la partie 1.2, suivie par nos contributions, exposées dans la partie 1.3. Une explication du plan de ce mémoire se trouve dans la partie 1.4.

1.1 Cadre d'étude

Dans la combinatoire, nos efforts de formalisation et de démonstration se concentrent sur certains objets, appelés cartes [Lando et al., 2004], et, dans une moindre mesure, sur une généralisation naturelle des cartes, les hypercartes.

Les cartes combinatoires sont des objets mathématiques qui apparaissent naturellement dans l'étude de nombreux problèmes, de nature mathématique ou physique [Planat et al., 2015, Diakite, 2015]. Les cartes dites étiquetées admettent une définition fondée sur deux permutations satisfaisant une certaine propriété de transitivité. Les cartes non étiquetées sont définies mathématiquement comme classes d'équivalence de cartes étiquetées. C'est ce type de carte qui est étudié par les mathématiciens et physiciens, et qui nous intéresse particulièrement. De par leur définition, les cartes non étiquetées sont cependant plus délicates à formaliser et à générer que les cartes étiquetées. Ces notions seront détaillées dans le chapitre 2.

Les cartes peuvent être formalisées de différentes manières, selon des approches directes ou bijectives, en faisant intervenir des structures telles que des permutations ou des mots. La formalisation des cartes combinatoires présente certaines difficultés, qui justifient l'intérêt de chercher des représentations plus élémentaires de ces objets.

Dans ce cadre, la spécification formelle, le test et la preuve de programmes sont appliquées à des implémentations d'algorithmes connus de combinatoire, mais également à des implémentations de nouveaux algorithmes, liés au cartes. L'accent est mis sur les algorithmes qui facilitent la génération ou le dénombrement des cartes, étiquetées ou non étiquetées.

Les preuves de programmes présentées dans cette thèse sont effectuées autant que possible automatiquement. Les programmes sont spécifiés en ACSL, pour une vérification avec le greffon WP de Frama-C (ce langage et cet outil sont présentés dans le chapitre 2). Cependant, tout en maintenant nos efforts d'automatisation des raisonnements, nous abordons aussi avec des preuves interactives les propriétés délicates à démontrer automatiquement. Selon l'approche adoptée et la propriété considérée, la part de preuve formelle sera plus ou moins étendue, et complétée par diverses validations par test pour renforcer la confiance dans les programmes proposés.

1.2 Problématique

Le premier objectif de cette thèse est d'illustrer comment l'utilisation de méthodes et d'outils de preuve de programmes peut être mis au service de la conception et la mise au point de programmes de génération de données structurées, avec un coût limité. En particulier, le présent travail illustre par différentes études de cas l'application de la spécification formelle, du test et de la vérification par preuve à des algorithmes, connus ou nouveaux, manipulant des tableaux d'entiers.

D'autre part, les prouveurs automatiques tels que les solveurs SMT évoluent rapidement et deviennent de plus en plus efficaces, si bien qu'il devient difficile d'établir précisément les capacités actuelles de ces outils. En effet, il s'avère qu'en pratique ces solveurs résolvent davantage de problèmes que les procédures de décisions qu'ils implémentent ne le laissent supposer. Face à ce constat, nous adoptons délibérément une approche em-

pirique consistant à soumettre à ces prouveurs automatiques des problèmes de plus en plus complexes, afin d'évaluer leurs capacités réelles.

La plupart des propriétés vérifiées sont des formules du premier ordre avec égalité, tableaux et arithmétique linéaire sur des entiers. Cette logique est indécidable [Bradley et al., 2006]. Certains fragments décidables sont connus. Nous conjecturons que nos problèmes dépassent le cadre de ces fragments et nous collectons ces problèmes afin de déceler des besoins en nouvelles procédures de décision. Cependant, nous excluons du cadre de ce mémoire cette identification et la recherche de nouvelles procédures. En effet, cette mise en évidence est une tâche conséquente qui interférerait avec les objectifs initiaux dans le champ d'investigation choisi, lui-même assez large. Ainsi, le deuxième objectif de cette thèse est de collecter de nombreux exemples de succès de preuve automatique, afin de mieux comprendre le fonctionnement des solveurs SMT, identifier leurs limites, et motiver la recherche ultérieure de nouvelles procédures de décision pour étendre ces outils. Le domaine de la combinatoire énumérative est particulièrement bien adapté à cet objectif, car il fournit de nombreux algorithmes dont la complexité et la structure sont en adéquation avec capacités supposées des solveurs SMT actuels.

Notre troisième objectif est la conception de générateurs de cartes étiquetées et non étiquetées, permettant de valider des programmes manipulant des cartes et des conjectures sur des propriétés de ces objets combinatoires.

Ces objectifs sont liés à certaines interrogations plus globales, parmi lesquelles on trouve :

- (Q_1) Face à quels types de problèmes peut-on espérer mener à bien une preuve formelle automatique utilisant des solveurs SMT ?
- (Q_2) Quels compromis, quelles adaptations doit-on faire sur le code pour parvenir à prouver automatiquement sa correction ?
- (Q_3) Dans le cas où le coût de la preuve devient trop important, quels types de tests effectuer ?
- (Q_4) Quels types de problèmes nécessitent une preuve interactive utilisant des assistants de preuve ?

Les réponses à ces questions dépendent de la manière de formaliser les structures considérées (telles qu'une permutation, un mot, une carte, etc.). Selon le choix de formalisation, il sera plus ou moins aisé de prouver ou tester certaines de leurs propriétés. Il faut ajouter à cela les problèmes d'adéquation entre la formalisation considérée et l'outil de vérification utilisé. En effet, les limites actuelles de certains outils de preuve formelle automatique ne permettent pas d'effectuer des raisonnements utilisant l'induction par exemple, ce qui nous oriente naturellement, dans ce cas précis, vers l'utilisation d'outils de preuve interactive, voire de test.

1.3 CONTRIBUTIONS

Nous avons appliqué différentes méthodes formelles à la combinatoire énumérative. Ce travail a débouché sur une expérience acquise dans le domaine de la preuve de programmes, mettant en lumière certaines limitations mais aussi certaines bonnes pratiques à mettre en œuvre. Nos contributions sont résumées dans [Genestier, 2016] et dans cette

partie. Nous présentons en premier lieu les apports au génie logiciel, puis les apports à la combinatoire énumérative. Cependant, cette répartition est un peu arbitraire, car la plupart de nos travaux sont des apports à ces deux domaines. C'est particulièrement le cas pour notre formalisation des permutations et nos preuves formelles de propriétés d'opérations combinatoires sur les permutations [Genestier et al., 2016], présentées dans le chapitre 4

1.3.1 CONTRIBUTIONS AU GÉNIE LOGICIEL

Dans le domaine du génie logiciel, nous avons conçu des générateurs exhaustifs bornés de tableaux structurés, en C, formellement spécifiés et vérifiés [Genestier et al., 2015a, Genestier et al., 2015b]. Nous fournissons également des patrons généraux d'aide à l'écriture de ces générateurs, des générateurs génériques et des preuves formelles automatiques de certaines de leurs propriétés.

Nous distribuons une bibliothèque *open source* contenant tous les programmes décrits dans ce mémoire, sous forme d'une archive `enum.*.tar.gz` téléchargeable depuis la page http://members.femto-st.fr/richard-genestier/en. Cette bibliothèque, dont ce mémoire sert de documentation, constitue un outil facilement utilisable par tout informaticien ou combinatoricien. Elle peut être utilisée pour générer simplement des objets grâce aux générateurs fournis, mais également pour concevoir de nouveaux générateurs par instanciation des patrons. D'autre part, ces générateurs vérifiés sont des composants de base pour tester d'autres algorithmes que ceux proposés dans la bibliothèque. Ils jouent alors le rôle d'outils certifiés de test. La plupart des générateurs de tableaux structurés de cette bibliothèque sont présentés dans le chapitre 3.

Nous avons également conçu des générateurs séquentiels vérifiés de tableaux codant les mots de Dyck, de couples de tableaux codant les mélanges de deux mots de Dyck, et de couples de tableaux codant les cartes planaires enracinées. Ils sont tous présentés dans le chapitre 6.

Comme contribution aux méthodes du génie logiciel, nous proposons une méthodologie originale combinant test exhaustif borné, test aléatoire et preuve formelle avec l'assistant de preuve Coq [Dubois et al., 2016]. Cette méthodologie est présentée dans le chapitre 8.

1.3.2 CONTRIBUTIONS À LA COMBINATOIRE ÉNUMÉRATIVE

Nous apportons également des contributions en combinatoire énumérative. La première est un algorithme générant séquentiellement les involutions sans point fixe d'une taille donnée. Une implémentation de cet algorithme est présentée dans le chapitre 5.

D'autre part, nous avons mené une étude combinatoire de classes d'équivalence de mots en bijection avec les cartes planaires [Baril et al., 2016]. Cette étude est présentée dans le chapitre 7.

Dans le domaine des cartes, nous proposons la notion originale de carte locale, codée par une seule permutation. Cette notion est présentée dans le chapitre 5 et approfondie dans les chapitres 8 et 9.

Des générateurs de cartes et d'hypercartes, basés sur des décompositions connues ou

originales, sont également exposés, dans les chapitres 9 et 10.

1.4 PLAN DU MÉMOIRE

Ce travail portant sur les domaines de recherche de la combinatoire et du génie logiciel, nous présentons dans le chapitre 2 les notions, méthodes et outils de ces deux domaines, nécessaires à la compréhension de la suite de ce mémoire. Ce chapitre présente également un état de l'art sur la formalisation des permutations et des cartes, objets centraux dans cette étude.

L'objectif d'effectuer des preuves formelles automatiques sur des objets complexes comme les cartes non étiquetées est assez ambitieux. Des approches préparatoires ont été effectuées pour en évaluer la faisabilité. Nous construisons d'abord des générateurs qui serviront de briques de base pour la génération de familles de structures combinatoires plus complexes, comme les cartes. Ce premier travail préparatoire est présenté dans le chapitre 3. Nous avons porté notre attention sur les algorithmes d'énumération combinatoire, nommés générateurs séquentiels, qui génèrent séquentiellement tous les tableaux structurés de même longueur, selon un ordre total prédéfini. Nous présentons une approche uniforme pour la mise en œuvre rationnelle de ces générateurs séquentiels de tableaux structurés. Nous vérifions alors automatiquement différentes propriétés [Genestier et al., 2015a, Genestier et al., 2015b].

Un deuxième travail préparatoire est présenté dans le chapitre 4. Les définitions des cartes combinatoires sont basées sur la notion de permutation. Nous proposons donc une formalisation en C des permutations et de différentes opérations sur les permutations qui seront utilisées dans la suite pour construire des cartes [Genestier et al., 2016].

Nous entrons dans la thématique des cartes à partir du chapitre 5. Avant d'aborder la génération des cartes non étiquetées, nous nous intéressons à celle de différents types de cartes étiquetées, ce qui constitue un objectif plus modeste. Ces objets étant des couples de permutations agissant transitivement sur un ensemble fini, nous avons conçu un algorithme caractérisant l'action transitive de deux permutations, et défini un générateur générique de couples de tableaux transitifs. Nous l'avons ensuite instancié pour obtenir divers générateurs séquentiels de familles de cartes étiquetées.

Nous commençons dans le chapitre 6 l'étude de la génération des cartes non étiquetées par un moyen détourné, sans utiliser leur définition mathématique, mais à travers l'une de leurs représentations par des mots. Nous réutilisons ici les patrons généraux d'aide à l'écriture de générateurs séquentiels définis dans le chapitre 3, afin de concevoir différents générateurs séquentiels de mots, dont un générateur séquentiel de couples de tableaux codant les mots en bijection avec les cartes planaires non étiquetées.

Ces générateurs nous ont permis d'étudier les positions de certains motifs dans ces mots et de dénombrer les classes de mots équivalents par les positions de leurs motifs [Baril et al., 2016]. Cette étude est présentée dans le chapitre 7.

Les limites de la preuve automatique auxquelles nous sommes confrontés dans le chapitre 5 sur les cartes étiquetées nous encouragent à considérer des preuves interactives. Dans la perspective de la génération des cartes non étiquetées, nous nous sommes intéressés dans le chapitre 8 à une génération de cartes basée sur des constructions inductives. Nous menons à cet effet une étude expérimentale, proposant une nouvelle mé-

thodologie basée sur des étapes préliminaires de test exhaustif borné et de test aléatoire, avant d'entamer les étapes de spécification et de preuve interactive. Dans ce contexte, nous reprenons la notion de carte locale définie dans le chapitre 5, ainsi que certaines opérations présentées dans le chapitre 4, pour concevoir deux opérations de construction de cartes. Nous avons formalisé cette notion, ces opérations et leur correction en Coq. Cette correction est validée par test exhaustif borné et aléatoire, avant d'être prouvée interactivement en Coq [Dubois et al., 2016].

Dans le chapitre 9, nous avons ensuite implémenté en C et spécifié formellement les opérations de construction de cartes locales présentées dans le chapitre 8, et prouvé automatiquement leur correction. Cette approche nous a permis d'étendre ce travail aux cartes planaires. Nous avons implémenté et spécifié des opérations de construction de cartes locales planaires, puis prouvé certaines de leurs propriétés. Une variante de ces opérations de construction a été conçue, sur laquelle nous avons effectué les mêmes vérifications. Différentes validations effectuées sur ces opérations nous permettent également d'établir des théorèmes de décomposition de cartes non étiquetées (planaires ou quelconques), et d'obtenir des générateurs de ces objets.

Dans le chapitre 10, nous considérons des opérations permettant de construire une hypercarte non étiquetée à partir de toute permutation connectée. Nous implémentons en C et spécifions ces opérations, nous montrons qu'elles produisent des permutations, et nous validons que ces permutations encodent des hypercartes non étiquetées. Une génération exhaustive bornée de permutations connectées nous permet alors d'obtenir un générateur d'hypercartes non étiquetées.

Le chapitre 11 effectue une synthèse de ce travail et donne des perspectives.

2

Contexte et état de l'art

Ce travail se situe à la frontière entre deux domaines de recherche : la combinatoire énumérative et le génie logiciel. Nous présentons dans ce chapitre les notions essentielles à la bonne compréhension de ce mémoire, pour les deux domaines de recherche considérés, ainsi qu'un état de l'art ciblé sur la formalisation des objets considérés. Le champ d'investigation étant assez vaste, certains chapitres ultérieurs peuvent également comporter des définitions et un état de l'art limités au sujet qu'ils traitent.

La partie 2.1 présente la combinatoire énumérative, ainsi que certaines de ses méthodes et outils utilisés dans ce mémoire. Une attention particulière y est portée sur les permutations et les cartes, objets combinatoires au coeur de ce travail. Les méthodes et outils du génie logiciel utilisés sont décrits dans la partie 2.2. La partie 2.3 présente un état de l'art sur la formalisation des permutations et des cartes. Enfin, certaines conventions et notations utilisées dans ce mémoire sont données dans la partie 2.4.

2.1 Eléments de combinatoire énumérative

Commençons par rappeler brièvement ce qu'est la combinatoire. On désigne par mathématiques discrètes la partie des mathématiques qui regroupe l'étude des structures discrètes, par opposition aux structures continues. Les objets étudiés en mathématiques discrètes sont des ensembles dénombrables, comme celui des entiers. La combinatoire est la partie des mathématiques discrètes qui étudie des ensembles finis ou dénombrables d'objets appelés structures combinatoires. Les arbres, les graphes, les permutations de n objets distinguables et les permutations particulières, telles que les involutions ou les dérangements, sont des exemples de structures combinatoires. D'autres structures combinatoires seront abordées dans le chapitre 3.

Lorsqu'on peut associer un entier naturel (appelé “taille”, “longueur”, “hauteur”, etc) à toutes les structures d'une famille dénombrable, de telle sorte que, pour chaque taille, les structures de cette taille soient en nombre fini, on dit que cette famille est énumérable. Dans ce cas, la combinatoire énumérative étudie le nombre de structures de chaque taille, en cherchant des moyens de calculer ces nombres ou des relations entre ces nombres, et à mettre au point des algorithmes efficaces pour les générer. La combinatoire bijective, quant à elle, cherche à exhiber des bijections non triviales entre familles de structures combinatoires.

La notion de série génératrice, permettant d'exhiber une formule pour compter le nombre de structures de chaque taille, est exposée dans la partie 2.1.1. De très nombreuses

séquences de nombres de structures de chaque taille sont répertoriées dans une encyclopédie en ligne nommée OEIS. Cet outil est détaillé dans la partie 2.1.2. Des notions de base sur les permutations et les cartes sont ensuite exposées dans les parties 2.1.3 et 2.1.4.

2.1.1 SÉRIE GÉNÉRATRICE

Une série génératrice est un outil algébrique qui permet de reformuler un problème de combinatoire énumérative sous la forme d'un problème de manipulation d'expressions algébriques.

En particulier, un problème classique de combinatoire énumérative est de déterminer le nombre a_n de structures d'un certain type qui sont de taille n. Il s'agit d'établir des propriétés de la suite $(a_n)_{n\geq 0}$. La série génératrice (ordinaire) associée à la suite $(a_n)_{n\geq 0}$ est la série formelle

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots = \sum_{n\geq 0} a_n x^n.$$

Cette série génératrice est une "somme infinie" formelle dont les coefficients sont les nombres de la suite $(a_n)_{n\geq 0}$. C'est aussi une fonction A de la variable formelle x, si bien qu'une série génératrice se nomme aussi fonction génératrice. Le nombre a_n est le n-ième coefficient du développement en série de Taylor de cette fonction A au voisinage de zéro.

Une analyse de la structure combinatoire étudiée permet souvent d'établir une équation vérifiée par sa série génératrice, appelée équation fonctionnelle. Résoudre cette équation permet d'obtenir une expression de cette série génératrice $A(x)$ en fonction de x, appelée forme close. On peut dériver a_n de cette forme close. Dans certains cas, certaines de ces tâches peuvent être effectuées par un logiciel de calcul formel.

Exemple 1. *Les mots de Dyck sur l'alphabet $\{a, \bar{a}\}$ (présentés en détail dans la partie 6.2.1 du chapitre 6) sont générés par la grammaire $S \to \varepsilon \mid aS\bar{a}S$ (où ε est le mot vide). Soit c_n le nombre de mots de Dyck de longueur $2n \geq 0$. On déduit de la grammaire l'équation fonctionnelle $C(x) = 1 + xC(x)^2$ vérifiée par la série génératrice $C(x) = \sum_{n\geq 0} c_n x^n$ de ces mots. La résolution de cette équation donne la forme close*

$$C(x) = \frac{1 - \sqrt{1-4x}}{2}.$$

Un développement en série de Taylor de cette forme close donne la formule

$$c_n = \frac{(2n)!}{(n+1)!n!}$$

pour ces nombres, dits de Catalan.

2.1.2 OEIS

L'encyclopédie en ligne des suites de nombres entiers [The OEIS Foundation Inc., 2010] (OEIS, pour *On-Line Encyclopedia of Integer Sequences*) est un site web permettant

d'effectuer librement des recherches dans une base de données de suites d'entiers présentant un intérêt mathématique. Elle a été fondée par le mathématicien N. Sloane, qui a commencé à collectionner les suites entières lorsqu'il était étudiant en 1960, pour soutenir son travail en combinatoire. L'OEIS est désormais la principale référence dans le domaine des suites d'entiers pour les mathématiciens professionnels et amateurs.

La base de données de l'OEIS contient plus de 250 000 suites, chacune possédant un identifiant unique. Elle est entièrement accessible, via une interface web, par un moteur de recherche : on peut rechercher une suite par sous-suite, par mot-clé, ou par identifiant. Chaque entrée propose les premiers termes de chaque suite, une ou des définitions, des références à des suites liées ou analogues, des liens vers la littérature, des formules, des séries génératrices, etc. Par exemple, les nombres de Catalan correspondent à la séquence A000108 accessible à l'adresse https://oeis.org/A000108.

Ainsi, lorsqu'un combinatoricien génère de nouvelles structures et les compte pour différentes tailles, il peut immédiatement savoir si ces objets sont comptés par une séquence non encore répertoriée, ou bien par une séquence existante. Dans le premier cas, il peut proposer l'ajout de sa séquence dans la base. Dans le second cas, il peut envisager de chercher une bijection structurelle entre ses structures et celles qui sont comptées par cette séquence de l'OEIS.

Lors de la conception d'un générateur de structures combinatoires répertoriées dans l'OEIS, un développeur peut immédiatement savoir si le nombre d'objets générés d'une certaine taille est correct, en se référant à la séquence correspondante de l'OEIS. Cette encyclopédie joue ainsi un rôle de validation.

2.1.3 Permutations

Les permutations sur un ensemble fini forment une famille combinatoire élémentaire mais centrale, largement étudiée en combinatoire énumérative. Nous exposons dans la partie 2.1.3.1 des notions de base sur les permutations, ainsi que les différentes notations et représentations utilisées dans ce mémoire. La partie 2.1.3.2 donne les définitions de certaines permutations particulières utiles pour la suite. Le lecteur désirant approfondir ces notions pourra se référer par exemple à [Stanley, 1997].

2.1.3.1 Définitions et notations

Définition 1 : Permutation

Une permutation p est une bijection sur un ensemble fini, identifié ici à l'ensemble $\{0, \dots, n-1\}$ des n premiers entiers naturels. Cet ensemble est appelé domaine ou support de la permutation p. La taille d'une permutation est la cardinalité n de cet ensemble.

Usuellement, les permutations sont définies sur l'ensemble $\{1, \dots, n\}$. Nous utilisons ici l'ensemble $\{0, \dots, n-1\}$ pour simplifier les explications des différents codes C agissant sur les permutations codées par des tableaux de longueur n, dont les indices commencent à 0.

L'image de $i \in \{0, \dots, n-1\}$ par une permutation p de taille n est notée $p(i)$.

Définition 2 : Composition de deux permutations

Soient p_1 et p_2 deux permutations de même support. On appelle composition ou produit de p_1 et p_2, et on désigne par $p_1\ p_2$, l'opération qui consiste à appliquer la permutation p_1 puis la permutation p_2.

La composition est ici appliquée de gauche à droite :

$$\forall\ i \in \{0, \ldots, n-1\},\ (p_1\ p_2)(i) = p_2(p_1(i)).$$

L'ensemble des permutations sur l'ensemble $\{0, \ldots, n-1\}$ muni de l'opération interne de composition est appelé groupe symétrique et est noté S_n. Le nombre de permutations de S_n est égal à $n! = 1 \times 2 \times \ldots n$ (séquence A000142 de l'OEIS).

On peut représenter toute permutation p de S_n sur deux lignes, par la matrice

$$\begin{pmatrix} 0 & 1 & \ldots & i & \ldots & n-1 \\ p(0) & p(1) & \ldots & p(i) & \ldots & p(n-1) \end{pmatrix}$$

ou sur une ligne par le mot $p(0)\ p(1)\ \ldots\ p(n-1)$ sur l'alphabet $\{0, \ldots, n-1\}$.

La permutation p est codée en C par le tableau `p` de longueur n tel que `p[`i`]` $= p(i)$ pour $0 \leq i \leq n-1$, qu'on peut représenter par | `p[0]` | `p[1]` | ... | `p[i]` | ... | `p[n-1]` |. Ce tableau est appelé représentation fonctionnelle de p. On notera `p[i..j]` la partie de la représentation fonctionnelle de p située entre les indices i et j inclus. Notons que nous utilisons également ces notations pour toute fonction sur les entiers à partir du chapitre 3.

Définition 3 : Cycle [Comtet, 1970]

Soient $x_1, x_2, \ldots, x_k$ k éléments distincts de $\{0, \ldots, n-1\}$ $(1 \leq k \leq n)$. Le cycle noté $(x_1\ x_2\ \ldots\ x_k)$ est la permutation $p \in S_n$ telle que $p(x_1) = x_2$, $p(x_2) = x_3, \ldots,$ $p(x_{k-1}) = x_k$, $p(x_k) = x_1$, et $p(x) = x$ si $x \neq x_i$.

En tant que permutation, p est de taille n. Par abus, en tant que cycle, on dit que p est de taille k. La notation $(x_1\ x_2\ \ldots\ x_k)$ est appelée notation cyclique de ce cycle. Lorsque $k = n$, on dit que p est une permutation circulaire. On dit que deux cycles sont disjoints s'ils ne modifient pas les mêmes éléments.

Proposition [1] (Décomposition cyclique)**.** *Toute permutation p de S_n peut être obtenue par produit de cycles disjoints de S_n, appelé décomposition (cyclique) de p.*

Parmi ces décompositions, la décomposition canonique de p est le produit de cycles dans lequel la notation de chaque cycle commence par son plus petit élément et les cycles sont écrits dans l'ordre croissant de leur plus petit élément. Les décompositions cycliques présentées dans ce mémoire seront données sous leur forme canonique.

On appelle cycle d'une permutation p tout cycle qui apparaît dans la décomposition cyclique de p. Un cycle de taille 1 de p est appelé point fixe de p.

Définition 4 : Conjugaison

La permutation conjuguée p' d'une permutation p par une permutation θ de même support est $p' = \theta^{-1}\ p\ \theta$. On dit alors que p' est le résultat de l'action de θ sur p par conjugaison.

Exemple [2]. *La permutation* $p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 0 & 4 & 2 \end{pmatrix} = 1\ 3\ 5\ 0\ 4\ 2 \in S_6$ *est codée par le tableau C* | 1 | 3 | 5 | 0 | 4 | 2 |. *Sa décomposition canonique est* $(0\ 1\ 3)\ (2\ 5)\ (4)$. *L'entier* 4 *est son unique point fixe.*

Soit la permutation $\theta = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 0 & 2 & 1 \end{pmatrix} = (0\ 3)\ (1\ 4\ 2\ 5) \in S_6$. *La permutation* θ^{-1} *est égale à* $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 4 & 0 & 1 & 2 \end{pmatrix} = (0\ 3)\ (1\ 5\ 2\ 4)$. *La composition* $\theta^{-1}\ p$ *est égale à* $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 2 & 4 & 1 & 3 & 5 \end{pmatrix} = (0)\ (1\ 2\ 4\ 3)\ (5)$, *et la permutation conjuguée* $\theta^{-1}\ p\ \theta$ *de* p *par* θ *est égale à* $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 2 & 4 & 0 & 1 \end{pmatrix} = (0\ 3\ 4)\ (1\ 5)\ (2)$.

2.1.3.2 Permutations particulières

La permutation $p \in S_n$ qui n'a que des points fixes (c.a.d. qui vérifie $\forall x \in S_n, p(x) = x$) est appelée identité de S_n et est notée Id_n. Une permutation $p \in S_n$ dont le produit par elle-même est égal à l'identité (c.a.d. qui vérifie $p^2 = Id_n$) est appelée involution. Les cycles d'une involution sont donc de taille un ou deux. Le nombre d'involutions de S_n est donné par la séquence $A000085(n)$ de l'OEIS [The OEIS Foundation Inc., 2010].

Une involution qui n'a que des cycles de taille 2 (c.a.d. qui vérifie $p^2 = Id_n \wedge \forall\ x \in S_n,\ p(x) \neq x$) ne possède pas de point fixe et est donc appelée involution sans point fixe. Ces involutions seront dénommées FFI par la suite, pour *Fixed-point Free Involution*. Toute FFI comporte nécessairement un nombre pair $n = 2k$ d'éléments et appartient donc au groupe symétrique S_{2k}. Le nombre de FFIs de S_{2k} est égal à la double factorielle $(2k-1)!! = 1.3.5\ldots(2k-1)$. Il est donné par la séquence A001147 de l'OEIS [The OEIS Foundation Inc., 2010].

Exemple [3]. *La permutation* $p_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 2 & 0 & 4 & 1 \end{pmatrix} = (0\ 3)\ (1\ 5)\ (2)\ (4) \in S_6$ *est une involution qui a pour points fixes* 2 *et* 4. *La permutation* $p_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 4 & 0 & 2 & 1 \end{pmatrix} = (0\ 3)\ (1\ 5)\ (2\ 4) \in S_6$ *est une involution sans point fixe.*

Un autre type de permutations particulières sera généré dans le chapitre 3 utilisé dans le chapitre 10.

Définition 5 : Permutation connectée [Cori et al., 2012]

Une permutation $p \in S_n$ est dite connectée (ou indécomposable) s'il n'existe pas d'entier $j < n$ tel que le préfixe $p[0..j-1]$ est une permutation de S_j.

Exemple [4]. *La permutation* $2\ 0\ 4\ 3\ 1 = (0\ 2\ 4\ 1)\ (3)$ *est une permutation connectée de* S_5, *tandis que* $2\ 0\ 1\ 4\ 3 = (0\ 2\ 1)\ (3\ 4)$ *n'est pas une permutation connectée de* S_5, *car son préfixe* $p[0..2] = 2\ 0\ 1 = (0\ 2\ 1)$ *est une permutation de* S_3.

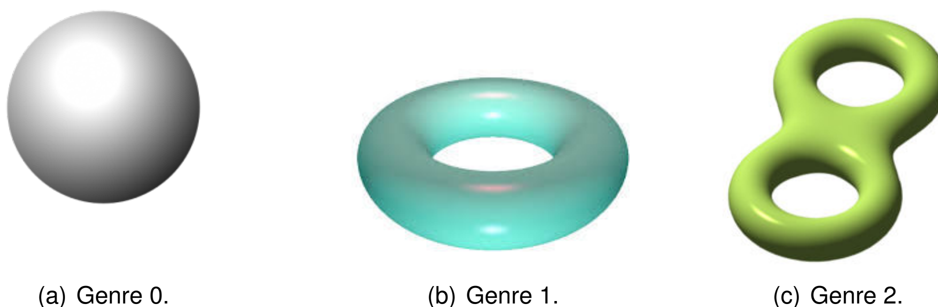
2.1.4 Cartes

Les cartes sont des objets mathématiques qui apparaissent naturellement dans l'étude de nombreux problèmes, de nature mathématique ou physique. L'objectif de cette partie est de présenter diverses définitions connues des cartes, et de déterminer parmi elles la plus "facile" à formaliser dans la suite de notre travail. Les notions de carte topologique, carte combinatoire étiquetée, et la correspondance entre ces deux notions sont respectivement exposées dans les parties 2.1.4.1, 2.1.4.2 et 2.1.4.3. Les notions de carte combinatoire enracinée (non étiquetée), carte générale et hypercarte sont respectivement exposées dans les parties 2.1.4.4, 2.1.4.5 et 2.1.4.6.

2.1.4.1 Cartes topologiques

Les cartes que nous allons considérer dans ce mémoire sont des cartes combinatoires, présentées dans les parties suivantes de ce chapitre. Cependant, on se sert constamment d'une représentation visuelle de ces objets, nommée carte topologique, afin de mieux percevoir certaines de leurs propriétés. L'objectif de cette partie est de présenter cette notion, de manière informelle, pour bien comprendre la correspondance avec les cartes combinatoires, exposée dans la partie 2.1.4.3.

La définition originelle d'une carte est de nature topologique et utilise le notion de surface (orientable) S_g de genre $g \geq 0$, qui désigne une sphère possédant g anses. La figure 2.1 présente une surface de genre 0 (2.1(a)), qui est une sphère, une surface de genre 1 (2.1(b)), nommée tore, ainsi qu'une surface de genre 2 (2.1(c)).



(a) Genre 0. (b) Genre 1. (c) Genre 2.

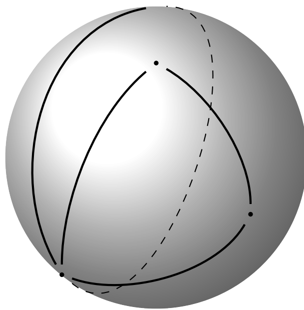
FIGURE 2.1 – Surfaces de genre 0, 1 et 2.

Une carte topologique de genre $g \geq 0$ est un graphe connexe (possédant potentiellement des boucles et des arêtes multiples) dessiné sur la surface S_g. Plus formellement :

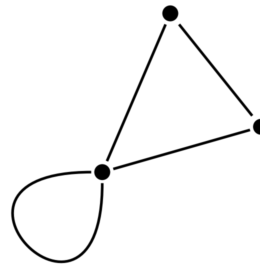
Définition 6 : Carte topologique [Lando et al., 2004]

Une carte (topologique) M sur une surface S_g est une partition de S_g en trois ensembles finis :

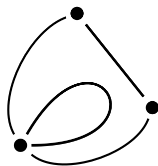
1. Un ensemble V de sommets de M, qui sont des points distincts de S_g,
2. un ensemble E d'arêtes de M, qui sont des courbes simples ouvertes de Jordan (mutuellement disjointes) dont les extrémités sont des sommets, et
3. un ensemble F des faces de M, qui sont les composantes connexes (mutuellement disjointes) du complément de $V \cup E$ dans S_g, chacune étant homéomorphe à un disque ouvert.



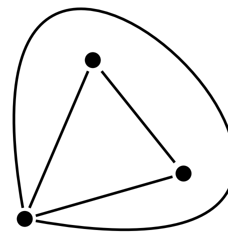
(a) Représentation sur une sphère.



(b) Représentation 1 sur le plan.



(c) Représentation 2 sur le plan.



(d) Représentation 3 sur le plan.

FIGURE 2.2 – Différentes représentations d'une carte planaire.

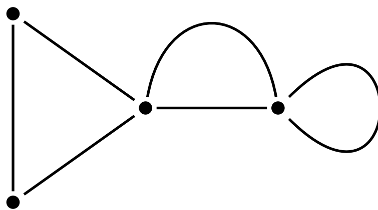
Nous admettons l'existence d'une carte topologique particulière, nommée carte vide (ou carte sommet), seulement constituée d'un sommet, sans arête, et n'ayant qu'une seule face. Toute autre carte topologique possède au moins une arête. Une carte possédant n arêtes sera également dite de taille n. Les cartes sont étudiées (mais aussi générées, comptées, etc.) à déformation continue de la surface près. Ainsi les isomorphismes de surface ne modifient pas les cartes.

Une carte planaire est une carte sur une surface de genre $g = 0$, c.a.d. une sphère. La figure 2.2(a) présente une carte planaire possédant 3 sommets, 4 arêtes et 3 faces, représentée sur une sphère. Par commodité, les cartes planaires sont représentées sur le plan, en associant une face à la partie infinie du plan. La même carte planaire peut

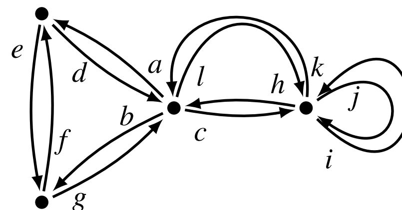
avoir plusieurs représentations sur le plan. Les figures 2.2(b), 2.2(c) et 2.2(d) présentent trois représentations possibles sur le plan de la carte planaire de la figure 2.2(a). Nous reviendrons sur cette question dans la partie 2.1.4.4.

Les arêtes des cartes topologiques peuvent être décomposées en deux arcs (ou brins) opposés. Une carte topologique peut alors être étiquetée par ses brins, comme le montre la figure 2.3(b). Par la suite, pour plus de commodité, nous identifierons un brin et son étiquette.

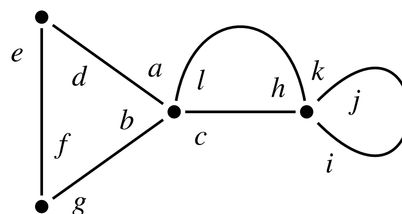
Un sommet et un brin sont dits incidents si le sommet est l'extrémité initiale du brin. Une boucle peut alors être définie comme une arête dont les deux brins sont incidents au même sommet. De même, on dit qu'un brin et une face sont incidents si la face borde le côté droit du brin lorsqu'on le parcourt de son extrémité initiale à son extrémité finale.



(a) Carte topologique.



(b) Carte topologique étiquetée avec arêtes décomposées en brins.



(c) Carte topologique étiquetée.

FIGURE 2.3 – Exemple de carte topologique étiquetée.

Le degré d'un sommet est égal au nombre total de brins incidents à ce sommet. Chaque face détermine un ordre cyclique sur ses brins incidents. Le degré d'une face compte le nombre total de brins incidents à cette face.

Pour la représentation des cartes dans ce mémoire, l'étiquette d'un brin est toujours placée dans sa face incidente et à proximité de son sommet incident.

Exemple [5]**.** *La figure 2.3 montre un exemple de carte topologique (2.3(a)) et d'une carte topologique étiquetée associée (2.3(b)). Dans cette dernière représentation, les brins h, i, j et k sont incidents au sommet le plus à droite. Le brin j est incident à la face située à l'intérieur de la boucle. Les 2 sommets de gauche sont de degré 2, tandis que le sommet central et celui de droite sont de degré 4. Cette carte a 4 faces, respectivement de degrés 6, 3, 2 et 1. La face de degré 6 est infinie dans cette représentation, tandis que celle de degré 1 est la face située à l'intérieur de la boucle. Dans cet exemple, les arêtes ont été représentées par deux brins distincts. Par la suite, pour simplifier la représentation, les deux brins seront représentés sur la même arête, comme le montre la figure 2.3(c).*

Cette définition topologique des cartes a l'avantage d'offrir une visualisation permettant de mener des raisonnements sur ces objets, mais elle est complexe à formaliser. En effet, cela nécessiterait la formalisation d'une partie importante de la topologie (surfaces, ouverts, continuité, courbes, homéomorphismes, etc) et nous entraînerait dans des domaines éloignés de nos objectifs. On préfère considérer une autre définition existante des cartes, celle de carte combinatoire présentée dans la suite de ce chapitre.

2.1.4.2 Cartes combinatoires étiquetées

L'étiquetage des cartes permet d'introduire la notion de carte combinatoire, indépendante de la topologie. Parmi les différentes familles de cartes combinatoires étiquetées, nous nous focalisons dans un premier temps sur les cartes combinatoires ordinaires étiquetées.

La définition d'une carte combinatoire ordinaire étiquetée utilise la notion d'action transitive d'un groupe de permutations.

Définition 7 : Action transitive

Un groupe de permutations G agit transitivement sur une partie X de leur support si et seulement si deux éléments quelconques de X peuvent être envoyés l'un sur l'autre par l'action d'une permutation du groupe. Formellement, $\forall x, y \in X \quad \exists g \in G \quad y = g(x)$.

En d'autres termes, cela signifie que tous les éléments de X sont "reliés" entre eux par les permutations du groupe G. Une carte combinatoire ordinaire étiquetée est alors définie de la manière suivante :

Définition 8 : Carte combinatoire ordinaire étiquetée

Une carte combinatoire ordinaire étiquetée est un triplet (D, R, L) où
- D est un ensemble fini à $d = 2e$ éléments,
- R est une permutation de D et
- L est une involution sans point fixe de D, telles que
- le groupe $\langle R, L\rangle$ engendré par les permutations R et L agit transitivement sur D.

Par la suite, pour simplifier les dénominations, toute carte désignée comme "combinatoire étiquetée" ou "étiquetée" sera implicitement ordinaire.

2.1.4.3 Correspondance entre cartes combinatoires étiquetées et cartes topologiques étiquetées

Les cartes combinatoires étiquetées et les cartes topologiques étiquetées sont en correspondance bijective[1], comme suit. Soit (V, E, F) une carte topologique étiquetée. On associe à cette carte topologique étiquetée la carte combinatoire étiquetée (D, R, L) telle que :

— Les éléments de D sont les étiquettes des arcs qui orientent les arêtes de l'ensemble E de la carte topologique. Ces éléments sont assimilés aux brins de la carte.

— Chaque cycle de la permutation R énumère les brins incidents à un sommet de l'ensemble V de la carte topologique, dans l'ordre d'une rotation autour de ce sommet, dans le sens inverse des aiguilles d'une montre (sens direct).

— Chaque cycle de l'involution L énumère les deux brins qui composent chaque arête de E, en associant chaque brin à son brin opposé (chaque cycle de L est de taille 2).

Il en résulte que chaque cycle de LR énumère les brins incidents à une face de l'ensemble F de la carte topologique. Cette correspondance justifie de désigner par ensemble de brins l'ensemble D d'une carte combinatoire (D, R, L), et par rotation la permutation R de cette carte.

Exemple [6]. *La carte combinatoire étiquetée associée à la carte topologique étiquetée de la figure 2.3 est le triplet* (D, R, L)*, où :*

— *L'ensemble des brins est* $D = \{a, b, c, d, e, f, g, h, i, j, k, l\}$.

— *La permutation des sommets est* $R = (a\ b\ c\ l)\ (d\ e)\ (f\ g)\ (h\ i\ j\ k)$*, où chaque cycle donne les brins rencontrés lors une rotation autour d'un sommet.*

— *La permutation des brins est* $L = (a\ d)\ (e\ f)\ (g\ b)\ (c\ h)\ (i\ j)\ (k\ l)$*, où chaque cycle échange un brin et son opposé.*

Chaque cycle de la permutation $LR = (a\ e\ g\ c\ i\ k)\ (b\ f\ d)\ (l\ h)\ (j)$ *est composé des brins incidents à l'une des faces de cette carte.*

La notion topologique d'incidence brin-sommet (resp. brin-face) se traduit alors de la manière suivante : on dit qu'un élément de D et un cycle de R (resp. LR) sont incidents si cet élément appartient à ce cycle. De même, le degré d'un sommet (resp. d'une face) est égal à la taille du cycle de R (resp. LR) correspondant.

Exemple [7]. *Dans la figure 2.3, le cycle de* R *correspondant au sommet le plus à droite est* $(h\ i\ j\ k)$*, indiquant que les brins* h, i, j *et* k *sont incidents à ce sommet, dont le degré est égal à* 4*. De même, le cycle de* LR *correspondant à la face située à l'intérieur de la boucle est* (j)*, indiquant que seul le brin* j *est incident à cette face, dont le degré est égal à* 1*. Le cycle de* LR *correspondant à la face infinie est* $(a\ e\ g\ c\ i\ k)$*, indiquant que cette face est de degré* 6*.*

Dans la définition 8, la notion d'action transitive sur D du groupe engendré par la paire de permutations (R,L) peut se traduire de la manière suivante : tout élément de D, c.a.d. tout brin d'une carte combinatoire étiquetée, peut être envoyé sur tout autre brin de D par une suite finie d'applications des deux permutations R et L. Au niveau topologique, cela

1. Plus rigoureusement, la correspondance est bijective avec les classes de cartes topologiques étiquetées équivalentes à difféomorphisme près (déformation continue de leur surface). Pour simplifier, chacune de ces classes est identifiée à l'une des cartes topologiques étiquetées qu'elle contient.

signifie que le graphe correspondant à la carte topologique associée est connexe, c.a.d. qu'il existe un chemin permettant de relier deux sommets quelconques de ce graphe.

Exemple [8]. *Dans la figure 2.3, le brin f peut par exemple être relié au brin l par un chemin suivant les brins intermédiaires g, b et c. Ce chemin est obtenu par l'application successive des permutations R, L, R, puis R. Ces brins peuvent également être reliés par un chemin suivant les brins intermédiaires e, d et a, correspondant à l'application successive des permutations L, R, L, puis R^{-1}.*

2.1.4.4 Cartes combinatoires enracinées

Pour l'étude des propriétés des cartes, il suffit souvent d'ignorer leurs étiquettes, c'est-à-dire de considérer leurs classes d'équivalence modulo renommage de leurs étiquettes. Les cartes combinatoires non étiquetées admettent des symétries qui rendent leur étude difficile. C'est pourquoi on se limite aux renommages préservant l'une des étiquettes, choisie à l'avance. Ce brin distingué dans D est nommé racine. Nous définissons ainsi la notion d'isomorphisme enraciné.

Définition 9 : isomorphisme enraciné

On appelle isomorphisme enraciné entre deux cartes combinatoires étiquetées (D,R,L) et (D,R',L') toute bijection θ de D préservant la racine et transformant les permutations R et L en R' et L' par conjugaison.

Les classes de cartes isomorphes par un isomorphisme enraciné forment une partition de l'ensemble des cartes étiquetées. Chaque classe est appelée carte combinatoire enracinée.

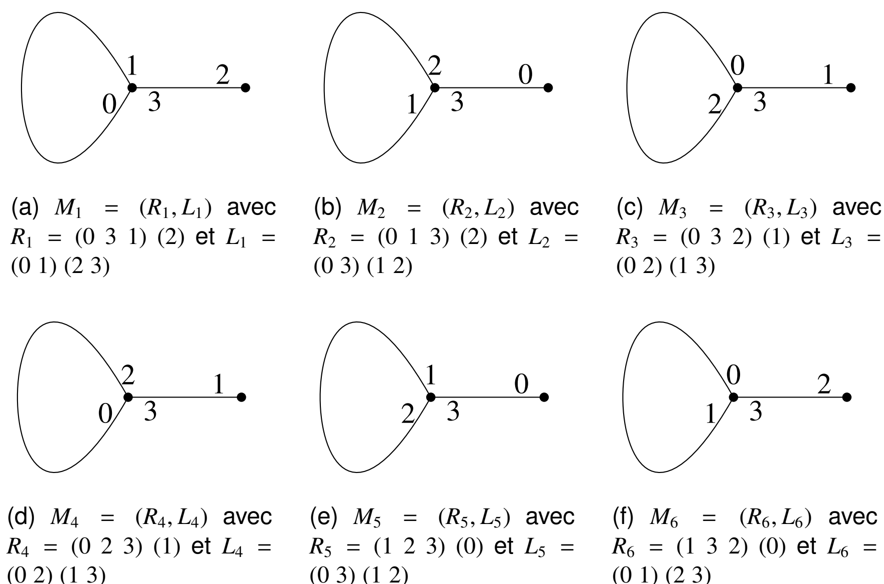
Définition 10 : Carte combinatoire enracinée

On appelle carte combinatoire enracinée (non étiquetée) toute classe d'équivalence de cartes combinatoires étiquetées pour la relation d'isomorphisme enraciné.

Une carte combinatoire enracinée "regroupe" toutes les cartes étiquetées qui deviennent identiques après renommage des étiquettes. C'est donc une carte combinatoire non étiquetée.

Notons que ce type de carte est souvent dénommé "pointée", mais, d'après [Guitter, 2015], "The map is pointed if it has a marked vertex (the pointed vertex) and rooted if it has a marked oriented edge (the root-edge)". Pour pouvoir distinguer ces deux sortes de cartes, nous traduisons ici "rooted map" par "carte enracinée" et non pas par "carte pointée".

L'ensemble D d'une carte (D,R,L) à e arêtes est quelconque. Cependant, nous pouvons, sans perte de généralité, identifier D à l'ensemble $\{0,\dots,2e-1\}$ pour permettre le codage machine des permutations par des tableaux d'entiers indexés de 0 à $2e-1$. D'autre part, nous verrons dans le chapitre 9 une décomposition des cartes combinatoires consistant à supprimer leur racine. Pour faciliter la formalisation de cette opération, nous choisissons comme racine de l'ensemble D son plus grand élément $2e-1$, car sa suppression des indices d'un tableau de longueur $2e$ indexé à partir de 0 donne encore un tableau indexé à partir de 0, de longueur $2e-1$.



(a) $M_1 = (R_1, L_1)$ avec $R_1 = (0\ 3\ 1)\ (2)$ et $L_1 = (0\ 1)\ (2\ 3)$

(b) $M_2 = (R_2, L_2)$ avec $R_2 = (0\ 1\ 3)\ (2)$ et $L_2 = (0\ 3)\ (1\ 2)$

(c) $M_3 = (R_3, L_3)$ avec $R_3 = (0\ 3\ 2)\ (1)$ et $L_3 = (0\ 2)\ (1\ 3)$

(d) $M_4 = (R_4, L_4)$ avec $R_4 = (0\ 2\ 3)\ (1)$ et $L_4 = (0\ 2)\ (1\ 3)$

(e) $M_5 = (R_5, L_5)$ avec $R_5 = (1\ 2\ 3)\ (0)$ et $L_5 = (0\ 3)\ (1\ 2)$

(f) $M_6 = (R_6, L_6)$ avec $R_6 = (1\ 3\ 2)\ (0)$ et $L_6 = (0\ 1)\ (2\ 3)$

FIGURE 2.4 – Six cartes combinatoires étiquetées appartenant à la même classe d'équivalence.

Exemple [9]. *La figure 2.4 montre six cartes combinatoires étiquetées à deux arêtes sur l'ensemble* $D = \{0, 1, 2, 3\}$*, dont la racine est le brin* 3*. L'isomorphisme enraciné* $\theta_1 = (0\ 1\ 2)\ (3)$ *transforme* R_1 *(resp.* L_1*) en* R_2 *(resp.* L_2*) par conjugaison. Les cartes* M_1 *et* M_2 *sont donc isomorphes et appartiennent à la même classe d'équivalence de cartes combinatoires étiquetées pour la relation d'isomorphisme enraciné. Il existe 4 autres isomorphismes enracinés* $\theta_2 = (0\ 2\ 1)\ (3)$*,* $\theta_3 = (0)\ (1\ 2)\ (3)$*,* $\theta_4 = (0\ 2)\ (1)\ (3)$ *et* $\theta_5 = (0\ 1)\ (2)\ (3)$ *différents de l'identité. A partir de la carte* M_1*, ces isomorphismes enracinés permettent d'obtenir respectivement les cartes* M_3*,* M_4*,* M_5 *et* M_6 *par conjugaison. Ces 6 cartes constituent donc une classe d'équivalence complète de cartes combinatoires étiquetées, c'est-à-dire une carte combinatoire enracinée.*

Une carte combinatoire enracinée sera désignée par la suite par l'acronyme ROM, pour *Rooted Ordinary Map*. Dans la représentation d'une ROM, nous faisons apparaître sa racine par une flèche sur le brin concerné, comme le montre la figure 2.5, qui présente les 10 ROMs possédant deux arêtes. Toutes sont planaires, sauf la carte 2.5(j) représentée sur un tore (surface de genre 1). Chacune des cartes enracinées de la figure 2.5 représente une classe d'équivalence de six cartes étiquetées. Par exemple, la classe d'équivalence constituée des six cartes étiquetées de la figure 2.4 correspond à la carte enracinée représentée dans la figure 2.5(e).

Le nombre de ROMs possédant e arêtes est donné par le $(e+1)$-ième élément $A000698(e+1)$ de la séquence A000698 de l'OEIS [The OEIS Foundation Inc., 2010], qui commence par 1, 2, 10, 74, 706, 8162, 110410, pour $0 \leq e \leq 6$. Chaque isomorphisme enraciné sur $\{0, ..., 2e-1\}$ renomme les $2e-1$ brins autres que la racine d'une carte combinatoire étiquetée (D, R, L) à e arêtes, pour donner une carte étiquetée différente. Puisqu'il existe $(2e-1)!$ isomorphismes enracinés sur $\{0, ..., 2e-1\}$, le nombre de cartes combinatoires étiquetées à e arêtes est égal au nombre de ROMs à e arêtes multiplié par $(2e-1)!$, soit $(2e-1)!\ A000698(e+1)$. Par exemple, à partir des 10 ROMs à 2 arêtes, on obtient $10 \times ((2 \times 2 - 1)!) = 10 \times 6 = 60$ cartes combinatoires étiquetées.

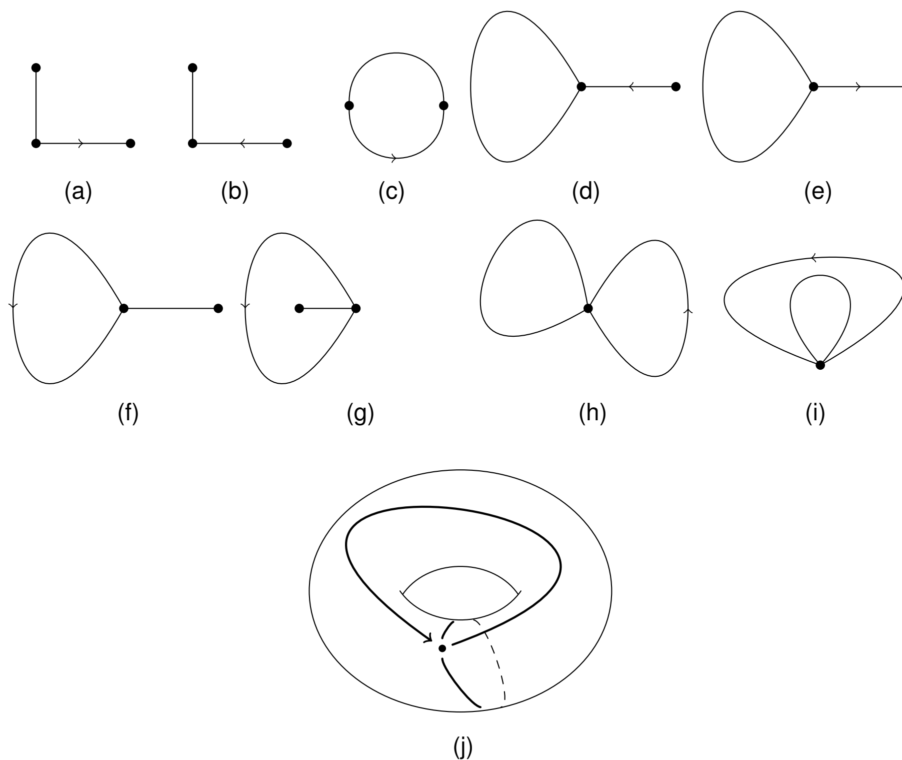


FIGURE 2.5 – Les 10 ROMs de taille 2.

Sur les cartes enracinées, nous pouvons distinguer deux faces particulières. La face extérieure d'une carte enracinée est la face incidente à sa racine. Sa face intérieure est la face incidente au brin opposé à sa racine. Nous notons alors $o(M)$ (resp. $i(M)$) le degré de la face extérieure (resp. intérieure) de la carte enracinée M, avec la convention que $i(M) = 0$ si la carte M n'a qu'une seule face.

Exemple 10. *Soit M_a (resp. M_f, M_i) la carte planaire 2.5(a) (resp. 2.5(f), 2.5(i)) de la figure 2.5. On a alors $o(M_a) = i(M_a) = 4$, $o(M_f) = 3$ et $i(M_f) = 1$, $o(M_i) = 1$ et $i(M_i) = 2$.*

Ces dernières notions seront utilisées dans le cadre de notre étude des cartes planaires, présentée dans le chapitre 9. Le problème de la non-unicité de la représentation d'une carte planaire dans le plan (évoqué dans la partie 2.1.4.1) peut se régler en adoptant la convention que la face infinie soit toujours la face extérieure. C'est le cas des neuf cartes planaires présentées dans la figure 2.5 et des des autres cartes planaires présentées dans ce mémoire.

2.1.4.5 CARTES GÉNÉRALES

Si l'involution L de la définition 8 peut avoir des points fixes, nous obtenons la notion de carte combinatoire générale.

Définition 11 : Carte combinatoire générale étiquetée

Une carte combinatoire générale étiquetée est un triplet (D, R, L) où
- D est un ensemble fini,
- R est une permutation de D et
- L est une involution de D, telles que
- le groupe $\langle R, L\rangle$ engendré par les permutations R et L agit transitivement sur D.

Notons que, pour une carte générale, l'ensemble D peut être de cardinalité d impaire. Une carte générale possédant d brins sera alors dite de taille d. Sur les cartes générales, on a la même notion d'isomorphisme enraciné que sur les cartes ordinaires, qui permet de définir la notion de carte générale enracinée (non étiquetée). La génération des cartes générales étiquetées sera abordée dans le chapitre 5.

La séquence $A140456$ de l'OEIS [The OEIS Foundation Inc., 2010] énumère le nombre $A140456(d)$ de cartes générales non étiquetées à d brins. Comme pour les cartes ordinaires, il existe $(d-1)!$ isomorphismes enracinés sur $\{0, ..., d-1\}$ renommant les $d-1$ brins autres que la racine d'une carte générale étiquetée (D, R, L) à d brins. Ainsi, le nombre de cartes générales étiquetées est $(d-1)!\ A140456(d)$.

2.1.4.6 Hypercartes

Si l'involution L de la définition 8 devient une permutation quelconque, nous obtenons la notion d'hypercarte combinatoire.

Définition 12 : Hypercarte combinatoire étiquetée

Une hypercarte combinatoire étiquetée est un triplet (D, R, L) où
- D est un ensemble fini,
- R et L sont deux permutations de D, telles que
- le groupe $\langle R, L\rangle$ engendré par les permutations R et L agit transitivement sur D.

Certaines remarques concernant les cartes générales sont également valables pour les hypercartes : l'ensemble D peut être de cardinalité d impaire ; une hypercarte possédant d brins sera dite de taille d ; la notion d'isomorphisme enraciné s'étend aux hypercartes étiquetées, et il permet de définir la notion d'hypercarte enracinée (non étiquetée). Les hypercartes étiquetées seront abordées dans le chapitre 5, tandis que les hypercartes enracinées (non étiquetées) constitueront l'objet d'étude principal du chapitre 10.

La séquence $A003319$ de l'OEIS [The OEIS Foundation Inc., 2010] énumère le nombre $A003319(d+1)$ d'hypercartes non étiquetées à d brins. Comme pour les cartes ordinaires et générales, il existe $(d-1)!$ isomorphismes enracinés sur $\{0, ..., d-1\}$ renommant les $d-1$ brins autres que la racine d'une hypercarte étiquetée (D, R, L) à d brins. Ainsi, le nombre d'hypercartes étiquetées est $(d-1)!\ A003319(d+1)$.

Dans la suite de ce mémoire, on utilisera parfois le terme "(hyper)carte" pour désigner globalement un objet qui peut être une carte ordinaire, une carte générale ou une hypercarte.

2.2 MÉTHODES ET OUTILS DU GÉNIE LOGICIEL

Le génie logiciel est le domaine de l'informatique qui étudie les méthodes de travail et les bonnes pratiques des ingénieurs qui développent des logiciels. Schématiquement, trois phases essentielles jalonnent la conception d'un logiciel : la spécification, l'implémentation, et la vérification. Dans le cadre de ce mémoire, nous nous intéressons aux techniques de spécification et de vérification visant à accroître la confiance dans la correction d'un programme, en utilisant des méthodes formelles.

Les méthodes formelles permettent d'obtenir une très forte assurance de l'absence de bug dans les logiciels. Elles sont utilisées dans le développement des logiciels les plus critiques. Leur amélioration et l'élargissement de leurs champs d'application pratique sont la motivation de nombreuses recherches scientifiques en informatique. Nous en avons une illustration dans notre travail, dans lequel le champ d'application est la combinatoire énumérative, et plus précisément l'étude des cartes combinatoires.

Dans la partie 2.2.1, nous présentons le contexte de spécification de notre étude. Dans la partie 2.2.2, nous présentons différentes techniques de test. Les principes de la preuve de programmes sont présentés dans la partie 2.2.3. Différents outils de vérification de programmes sont présentés dans la partie 2.2.4, notamment ceux utilisés lors de cette étude.

2.2.1 PROGRAMMATION PAR CONTRAT

Les contrats sont apparus dans le langage Eiffel [Meyer et al., 1987], basé sur le paradigme de la programmation orienté objet. La programmation par contrat est un paradigme de programmation dans lequel le comportement d'un programme est décrit par des clauses formelles, afin de réduire le nombre de bugs dans les programmes [Liskov et al., 1986]. Le contrat précise ce qui doit être vrai à un moment donné de l'exécution d'un programme.

Nous utilisons une programmation par contrat mêlant spécification et implémentation dans un seul fichier. Les programmes que nous considérons sont constitués de quelques fonctions, proches des algorithmes qu'elles implémentent. Les deux phases de spécification et implémentation sont donc ici fusionnées.

Les langages décrivant une spécification formelle à travers des contrats sont appelés langages de contrat. Il en existe de nombreux, basés sur la logique du premier ordre. Certains sont dédiés à un langage de programmation, comme par exemple Spec# [Barnett et al., 2011] pour C#, ACSL [Baudin et al., 2013] et VCC [Dahlweid et al., 2009] pour C, JML [Ahrendt et al., 2014] pour Java, ou Praspel [Enderlin et al., 2011] pour PHP. D'autres ne sont pas liés à un langage de programmation pré-existant, mais sont intégrés dans un langage de programmation conçu pour la preuve, comme par exemple WhyML pour Why3 [Bobot et al., 2013] ou Dafny [Leino, 2010].

Pour rendre les méthodes de spécification et de preuve accessibles à un large public, nous avons choisi pour nos implémentations le paradigme de la programmation impérative, à travers l'usage du langage C. Créé au début des années 1970, ce langage est devenu un des langages les plus utilisés. De nombreux langages plus modernes, comme C++ ou Java par exemple, ont une syntaxe similaire à celle de C. En particulier, certains combinatoriciens connaissent et utilisent fréquemment ce langage. Les travaux que nous

présentons dans ce mémoire leur sont donc accessibles sans effort supplémentaire.

2.2.2 TEST

Nous pouvons utiliser des techniques de test, soit pour valider des programmes et des conjectures avant de tenter de les prouver formellement, soit pour accroître la confiance dans la correction d'un programme en cas d'échec de la preuve automatique.

Dans le cadre de nos travaux, nous avons utilisé deux méthodes de test : le test exhaustif borné, présenté dans la partie 2.2.2.1, et le test aléatoire, présenté dans la partie 2.2.2.2.

2.2.2.1 TEST EXHAUSTIF BORNÉ

Le test exhaustif borné (BET, pour *Bounded-Exhaustive Testing*) [Sullivan et al., 2004] est une méthode élémentaire pour la génération automatique de données de test. Il consiste à générer toutes les données possibles jusqu'à une taille donnée. Cette technique est intéressante, car d'après [Jackson et al., 1996], "a large portion of faults is likely to be revealed already by testing all inputs up to a small scope". De fait, cette méthode de test est particulièrement bien adaptée à la détection d'erreurs dans des programmes portant sur des structures de données, car elle fournit des contre-exemples de taille minimale [Senni et al., 2012], et les erreurs dans les fonctions à tester peuvent souvent être révélées grâce à des données d'entrée de petite taille. L'utiliser dans le cadre de la combinatoire énumérative est pertinent, car les structures combinatoires d'une taille donnée sont en nombre fini, par définition, et car la recherche en combinatoire propose de nombreux algorithmes de génération exhaustive bornée de structures combinatoires.

Une approche existante de ce type de test est le test basé sur le type (type-targeted testing [Seidel et al., 2014]), dans lequel les types sont convertis en requêtes pour des solveurs SMT dont les réponses fournissent des contre-exemples. SmallCheck et Lazy SmallCheck [Runciman et al., 2008] sont deux bibliothèques Haskell qui automatisent le BET de propriétés pour des programmes fonctionnels. Elles utilisent également des générateurs basés sur le type afin d'obtenir des ensembles de test à valeurs finies pour lesquelles les propriétés sont vérifiées, et de renvoyer tous les contre-exemples trouvés. Au lieu d'utiliser un échantillon de valeurs générées au hasard, elles testent les propriétés pour toutes les valeurs jusqu'à une certaine limite de profondeur, augmentant progressivement cette limite.

Une approche plus spécifique est le test basé sur des contrats (CBT, pour *Contract-Based Testing*), utilisant des langages de contrat. Pour les programmes Java, les outils TestEra [Marinov et al., 2001] et UDITA [Gligoric et al., 2010] se fondent sur le langage de spécification JML. Ces outils nécessitent que l'utilisateur écrive les méthodes de génération de données ainsi que les prédicats. Pour une taille donnée de structure en entrée, ces outils génèrent alors automatiquement toutes les données de test non isomorphes et évaluent les critères de correction. Si un critère de correction n'est pas vérifié, ils donnent des contre-exemples en Java.

Pour les programmes C spécifiés avec ACSL, l'outil StaDy [Petiot et al., 2014] intègre le générateur de test structurel PathCrawler [Williams, 2010] au sein de la plate-forme d'analyse statique Frama-C. PathCrawler utilise l'exécution concrète et l'exécution symbolique basée sur la résolution de contraintes et permet à StaDy de guider l'utilisateur

dans son travail de preuve, en montrant les incohérences entre le code et la spécification par la couverture de test de tous les chemins possibles du code et de la spécification, et en produisant des contre-exemples.

Dans le cadre de ce travail, nous utilisons le test basé sur des contrats (CBT), en ne générant que les structures qui satisfont des contraintes données. Cette technique, qui nécessite d'écrire nos propres générateurs, demande plus d'efforts, et se rapproche des outils TestEra et UDITA. Plus précisément, nous utilisons le BET par génération exhaustive bornée de structures combinatoires en C. Cette démarche est détaillée dans différents chapitres de ce mémoire, dont le chapitre 3. En complément, nous avons utilisé l'outil StaDy pendant une courte période, par l'intermédiaire de G. Petiot, qui possédait une licence PathCrawler. Nous effectuons également du BET par le biais de la programmation logique, comme détaillé dans la partie 8.1.3 du chapitre 8.

2.2.2.2 Test aléatoire

Le test aléatoire est une technique de test où les programmes sont testés par génération aléatoire de données. Les résultats sont comparés aux spécifications du programme pour vérifier si la sortie de test est conforme ou non. Le test aléatoire basé sur des propriétés (RPBT, pour *Randomized Property-Based Testing*) consiste à générer aléatoirement des données de test pour valider des propriétés portant sur des programmes. Le RPBT a beaucoup gagné en popularité depuis l'apparition de QuickCheck pour Haskell [Claessen et al., 2000], suivi entre autres par Quickcheck pour Isabelle [Bulwahn, 2012], ou encore une implémentation pour le langage C, quickcheck4c [Zito, 2014]. Dans le RPBT, un générateur de données aléatoires peut être défini par filtrage de la sortie d'un autre générateur, de la même manière qu'un générateur exhaustif peut être défini par filtrage d'un autre générateur exhaustif dans le BET.

Dans le cadre de notre étude, nous utilisons du RPBT pour tester des conjectures Coq avec le plugin QuickChick [Hritcu et al., 2015]. Cet outil est inspiré de QuickCheck pour Haskell. Cette démarche est détaillée dans la partie 8.1.2 du chapitre 8.

2.2.3 Preuve de programmes

Lorsqu'un programme est doté d'une spécification fonctionnelle, il est possible de démontrer qu'il correspond à sa spécification, à l'aide d'une méthode de preuve formelle fondée sur la logique de Hoare.

La logique de Hoare [Hoare, 1969] est une méthode formelle définie par le chercheur en informatique britannique T. Hoare. La correction d'un programme est décrite et démontrée par induction sur la structure du programme : à chaque règle syntaxique de construction d'un programme correspond une règle de la logique de Hoare.

Ces règles utilisent la notion de triplet de Hoare : $\{P\}\ \mathtt{I}\ \{Q\}$, où P et Q sont des prédicats et `I` est une instruction. Le prédicat P, nommé précondition, décrit un ensemble d'états en entrée de l'instruction `I`. Le prédicat Q, nommé postcondition, caractérise un ensemble d'états après transformation par l'instruction I. Un triplet de Hoare $\{P\}\mathtt{I}\{Q\}$ est valide si la condition suivante est vérifiée : si l'état du système vérifie P et si l'instruction `I` termine, alors l'état du système après exécution de `I` vérifie Q. Par exemple, le triplet

$$\{x = y\}\ x := x + y\ \{x = 2y\}$$

est valide.

Soit $\{P\}\mathtt{I}\{Q\}$ un triplet de Hoare valide. On appelle plus faible précondition de I par rapport à Q, et on note $WP(I,Q)$, le prédicat P tel que, pour toute formule P' rendant le triplet $\{P'\}\mathtt{I}\{Q\}$ valide, P' implique P. Pour prouver qu'un triplet de Hoare $\{P\}\mathtt{I}\{Q\}$ est valide, il suffit donc de montrer que P implique $WP(I,Q)$. Ce calcul se nomme calcul de plus faible précondition [Dijkstra, 1975] ("*Weakest Precondition Calculus*").

En pratique, pour prouver la correction d'un programme, on utilise un générateur de conditions de vérification, qui prend en entrée un programme annoté et retourne une liste de formules logiques, appelées conditions de vérification, qui impliquent la correction du programme.

Un exemple de générateur de conditions de vérification est le greffon WP [Baudin et al., 2015] (pour *Weakest Precondition*). Comme son nom l'indique, cet outil utilise le calcul de plus faible condition pour générer les conditions de vérification.

Il convient ensuite de démontrer mathématiquement ces formules. Ces démonstrations mathématiques sont des tâches techniques, souvent longues et fastidieuses. Elles peuvent être mécanisées grâce à des logiciels appelés prouveurs de théorèmes.

2.2.4 Outils de vérification

Il existe de nombreux logiciels d'aide à la vérification de programmes. Nous présentons dans la partie 2.2.4.1 les prouveurs automatiques utilisés dans nos expérimentations. La partie 2.2.4.2 décrit des plate-formes d'analyse de programmes. La plate-forme Frama-C, utilisée tout au long de notre étude, est détaillée dans la partie 2.2.4.3. La partie 2.2.4.4 décrit le principe des prouveurs interactifs, le prouveur Coq étant détaillé dans la partie 2.2.4.5.

2.2.4.1 Solveurs SMT

Les prouveurs automatiques utilisés dans notre travail pour décharger les conditions de vérification sont des solveurs SMT (pour "Satisfiability Modulo Theories"). Ces outils implémentent des procédures de décision pour des fragments logiques, dans des théories sur l'arithmétique, les tableaux, les types de données algébriques, etc. En preuve de programmes, ils cherchent à satisfaire la formule logique du premier ordre exprimant la condition de vérification. Ils sont entièrement automatiques, et une durée de calcul par défaut est allouée pour chaque formule logique traitée. Ils valident (ou déchargent) les conditions de vérification, ou échouent, soit parce qu'ils ne disposent pas d'assez de temps de calcul, soit parce que les procédures de décision qu'ils implémentent ne sont pas adaptées pour les décharger. De nombreux solveurs SMT existent, comme par exemple Z3 [De Moura et al., 2008], CVC3 [Barrett et al., 2007], CVC4 [CVC4, 2012], Alt-Ergo [Alt-ergo, 2006] ou Yices [Dutertre, 2014].

Ces outils peuvent être utilisés de façon autonome, ou bien au sein de plate-formes d'analyse de programmes.

2.2.4.2 PLATE-FORMES D'ANALYSE DE PROGRAMMES

Certaines plate-formes d'analyse de programmes automatisent la preuve de programmes via l'implémentation d'un générateur de conditions de vérification de programmes spécifiés. On trouve parmi elles KeY [Beckert et al., 2007], Why3 [Bobot et al., 2013] et Frama-C [Kirchner et al., 2012].

Les condition de vérification générées par le système KeY sont traitées par un solveur interne auquel on peut adjoindre certains solveurs SMT. Sa particularité réside dans le fait qu'en cas d'échec de la preuve automatique, il est possible d'ajouter manuellement des règles de déduction, nommées “taclets”, et de relancer la preuve automatique ensuite.

L'outil Why3 offre une large palette de possibilités, en ce sens que les conditions de vérification qu'il génère sont dans le format d'entrée de très nombreux solveurs SMT, mais aussi des assistants de preuve Coq, PVS et Isabelle/HOL. Son langage de programmation et de spécification, WhyML, est un dialecte ML.

Le choix de la programmation impérative fixé pour cette thèse, ainsi qu'un partenariat entre le Département d'Informatique des Systèmes Complexes de l'Université de Franche-Comté et le CEA LIST, nous ont encouragés à utiliser pour notre étude la plate-forme Frama-C.

2.2.4.3 FRAMA-C

Les algorithmes présentés dans cette étude seront implémentés en C et spécifiés en ACSL (pour *ANSI C Specification Language*) [Baudin et al., 2013]. ACSL est un langage dédié à l'analyse statique de programmes C qui permet de spécifier formellement des contrats qui doivent être vérifiés par le programme. Une telle spécification sera détaillée dans la partie 3.2.2 du chapitre 3.

Nous utilisons la plateforme d'analyse de programmes C Frama-C [Kirchner et al., 2012], développée par le CEA LIST et INRIA Saclay, qui utilise le langage de spécification ACSL. Pour la preuve de programmes, nous utilisons le greffon WP [Baudin et al., 2015], qui implémente un générateur de conditions de vérification pour les programmes C annotés en ACSL, en utilisant Why3 [Bobot et al., 2013]. Celles-ci peuvent être traitées par l'assistant de preuve Coq ou déchargées par des solveurs SMT. De nombreux solveurs peuvent être intégrés dans ce système, en plus du solveur SMT Alt-Ergo [Alt-ergo, 2006], présent par défaut. Pour notre étude, nous lui avons adjoint les solveurs CVC3 [Barrett et al., 2007] et CVC4 [CVC4, 2012].

Dans le cadre de leur utilisation avec Frama-C, les solveurs déchargent une condition de vérification générée en renvoyant le résultat "Valid" (assorti d'une durée de preuve), ou échouent en précisant qu'ils ne disposent pas d'assez de temps de calcul (et renvoient le résultat "Timeout") ou que les heuristiques qu'ils incorporent et les procédures de décision qu'ils implémentent ne leur permettent pas de prouver le but dans le temps imparti (et renvoient le résultat "Unknown"). La durée de calcul par défaut allouée à chaque condition de vérification est de 10 secondes. Il est possible d'augmenter cette durée dans le cas où certaines conditions de vérification complexes nécessitent davantage de temps pour être déchargées.

2.2.4.4 Prouveurs interactifs

D'autres systèmes, appelés prouveurs interactifs ou assistants de preuve, utilisent une logique non décidable ; l'action de l'utilisateur est alors nécessaire pour mener certaines preuves. Des exemples de tels systèmes sont Coq [Bertot et al., 2004], PVS [Owre et al., 1996], ACL2 [Kaufmann et al., 2004] et Isabelle [Paulson, 1993].

Une amélioration notoire dans ce domaine est l'introduction de prouveurs automatiques dans les assistants de preuve. Par exemple, Isabelle/HOL avec son outil Sledgehammer [Meng et al., 2008], et HOL4 qui intègre des solveurs SMT [Weber, 2011], épargnent à l'utilisateur de traiter certaines obligations de preuve.

Dans le cadre de cette étude, nous utilisons l'assistant de preuve Coq.

2.2.4.5 Coq

Coq [Bertot et al., 2004] est un assistant de preuve développé par l'équipe PI.R2 d'Inria. Coq est fondé sur le calcul des constructions inductives (CoC, pour *Calculus of Constructions*), une théorie des types d'ordre supérieur. Son langage de spécification, Gallina, est une forme de lambda-calcul typé. Coq peut être vu aussi bien comme un langage de programmation que comme un système de preuves mathématiques. Il permet de synthétiser des programmes certifiés à partir de preuves constructives de leurs spécifications.

Notre utilisation de l'assistant de preuve Coq est détaillée dans le chapitre 8.

2.3 Formalisations existantes

Les cartes combinatoires, objets liés aux permutations, sont très étudiées par les mathématiciens et les informaticiens. Plusieurs travaux de formalisation de ces objets existent. Les parties 2.3.1 et 2.3.2 présentent respectivement les travaux existants de formalisation des permutations et des cartes.

2.3.1 Formalisation des permutations

Les permutations, structures combinatoires très répandues, ont été formalisées dans une bibliothèque de l'assistant de preuve PVS. Une permutation y est définie comme un produit de transpositions et est représentée par une liste de ces transpositions [Ayala-Rincón et al., 2016]. Des formalisations des permutations se trouvent également dans différentes bibliothèques de l'assistant de preuve Coq. Le module Sorting de la bibliothèque standard Coq (8.4) propose une notion de permutation en tant que relation entre deux listes, définie de deux manières : à partir de transpositions [2], et par la condition que les deux listes contiennent les mêmes éléments avec la même multiplicité [3].

La bibliothèque Ssreflect pour Coq [Gonthier et al., 2015] propose une notion de permutation en tant que relation entre deux séquences [4], mais aussi en tant qu'endofonction

2. https://coq.inria.fr/library/Coq.Sorting.Permutation.html.
3. https://coq.inria.fr/library/Coq.Sorting.PermutSetoid.html.
4. http://ssr.msr-inria.inria.fr/~jenkins/current/mathcomp.ssreflect.seq.html.

injective sur un type fini [5]. De plus, une bibliothèque Ssreflect complémentaire [6], développée par F. Hivert, formalise une permutation sur $\{0, \ldots, n-1\}$ par un mot qui est sa représentation sur une ligne.

Toutes ces formalisations font partie de bibliothèques d'assistants de preuve et ne répondent donc pas à nos attentes d'automaticité de nos preuves. Nous allons donc créer notre propre formalisation des permutations adaptée à notre objectif. Nous ne reprenons pas l'idée de représenter les permutations par des listes car les solveurs SMT ne permettent pas de raisonner sur ces structures. Notre formalisation des permutations sera détaillée dans le chapitre 4.

2.3.2 Formalisation des cartes

La théorie des cartes combinatoires a été développée dans les années 70 par W. Tutte [Tutte, 1971, Tutte, 1979]. Il a élaboré une théorie axiomatique des cartes combinatoires sans faire référence à la topologie. Plus récemment, F. Lazarus [Lazarus, 2014] a mené une approche algorithmique sur les graphes et les surfaces basée sur les cartes combinatoires. L'implémentation des cartes combinatoires est un sujet actif de recherche depuis les années 90 [Lienhardt, 1991, Bertrand, 1998, Damiand et al., 2014, Bertrand et al., 1998], car ces objets sont au cœur de modeleurs géométriques à base topologique et de bibliothèques de géométrie algorithmique [Damiand, 2010, Kraemer et al., 2013].

Il existe dans la littérature diverses formalisations des cartes proposées pour prouver des théorèmes. Une formalisation avancée liée aux cartes est celle des hypercartes combinatoires, pour prouver le théorème des quatre couleurs en Coq [Gonthier, 2008, Gonthier, 2005]. Les hypercartes combinatoires généralisent les cartes combinatoires, comme détaillé dans la partie 2.1.4.6 du chapitre 2. Cette formalisation adopte la définition alternative d'une hypercarte comme un triplet d'endofonctions dont le produit est l'identité [Gonthier, 2005, p.19]. Sur un ensemble fini, cette condition est suffisante pour en déduire que les trois endofonctions sont des bijections.

Certains preuves formelles sur les cartes combinatoires (ou des variantes) ont été réalisées dans le domaine de la géométrie algorithmique. Dufourd et al. ont développé une bibliothèque Coq conséquente spécifiant les hypercartes, utilisée pour prouver certains résultats classiques tels que la formule d'Euler pour les polyèdres [Dufourd, 2008], le théorème de Jordan [Dufourd, 2009], ainsi que des algorithmes tels que celui de l'enveloppe convexe [Brun et al., 2012] et celui de la segmentation d'images [Dufourd, 2007]. Dans ces travaux, une carte ou une hypercarte combinatoire est représentée par un type inductif possédant des contraintes. Ses constructeurs procèdent par insertion d'un brin ou par liaison de deux brins. Dans [Dubois et al., 2007], C. Dubois et J.-M. Mota ont proposé une formalisation des cartes généralisées utilisant le formalisme B, très proche de la présentation mathématique avec des permutations et involutions.

De manière générale, la formalisation des cartes combinatoires présente certaines difficultés [Dufourd et al., 2000, Gonthier, 2005], qui justifient l'intérêt de chercher des représentations plus élémentaires de ces objets.

Dans le cadre de ce travail, nous abordons la génération de cartes et d'hypercartes,

5. http://ssr.msr-inria.inria.fr/~jenkins/current/mathcomp.fingroup.perm.html.
6. https://www.lri.fr/~hivert/Coq-Combi/.

étiquetées et non étiquetées, en gardant notre objectif d'automatisation de certaines preuves. Nous reprenons leurs définitions mathématiques basées sur des permutations. Nous proposons également une formalisation simplifiée des cartes ordinaires, à travers la notion de carte locale, détaillée dans le chapitre 5.

2.4 Conventions et notations

Les algorithmes décrits dans ce mémoire sont implémentés en C. Nous avons fait le choix de présenter ces algorithmes directement par des fonctions C richement commentées, plutôt que par du pseudo-code. En effet, nous présentons, lorsqu'elle existe, la spécification formelle de ces fonctions, afin d'effectuer la preuve formelle de leur correction. Dans un souci de lisibilité, nous prenons alors la convention d'adopter une notation mathématique dans le texte pour les variables des programmes, par exemple p_1 pour la variable `p1`. Le premier exemple de fonction C ainsi présentée et décrite se trouve dans la partie 3.2.2 du chapitre 3. En outre, pour faciliter la lecture des spécifications, certaines notations ACSL sont remplacées par des symboles mathématiques (par exemple, les mots-clés `\forall` et `integer` sont respectivement notés $\forall$ et $\mathbb{Z}$).

Toutes les statistiques de preuve et test présentées dans ce mémoire ont été effectuées sur un PC Intel Core i5-3230M à 2.60GHz $\times$ 4 sous Linux Ubuntu 14.04. Nous utilisons pour la preuve de programmes C Frama-C Neon 20140301 et son greffon WP 0.8 utilisant Why3 0.82 assisté des solveurs Alt-Ergo 0.95.2, CVC3 2.4.1 et CVC4 1.3.

3

Générateurs séquentiels de tableaux structurés

3.1 Introduction

A partir de ce chapitre, nous entrons pleinement dans les contributions de la thèse. Rappelons que l'objectif global de la thèse est d'effectuer des preuves formelles automatiques sur des objets complexes comme les cartes enracinées. Cet objectif étant assez ambitieux, nous avons adopté une approche par étapes, d'une part pour en évaluer la faisabilité, et d'autre part pour évaluer les limites des outils de vérification automatique utilisés. Ce chapitre présente la mise au point d'un socle sur lequel vont reposer les étapes suivantes. Il reprend et étend les communications [Genestier et al., 2015a] et [Genestier et al., 2015b].

Les cartes enracinées étant des structures combinatoires définies à partir de permutations, nous nous sommes tout d'abord intéressés à des structures combinatoires plus élémentaires, en particulier celles qui sont liées aux permutations (injections, surjections). Chaque structure est encodée en C par un tableau satisfaisant certaines propriétés. Nous avons étudié divers algorithmes permettant de générer ces tableaux un par un. Après avoir implémenté, spécifié et vérifié formellement des algorithmes issus de la littérature, nous avons généralisé leurs principes de génération, afin de faciliter leur vérification déductive et préparer ainsi nos outils et méthodes à l'objectif final.

Une tendance récente de recherche en vérification du logiciel vise à la construction d'environnements de vérification qui sont eux-mêmes certifiés, afin d'éviter de valider à tort des propriétés de logiciels critiques utilisés dans le domaine de la sécurité. En particulier, un défi pour le test unitaire de génération de données structurées est de concevoir et d'implémenter des générateurs de structures de données complexes certifiés. Par exemple, une méthode de test aléatoire nommée Randomized property-based testing a été formalisée en Coq pour certifier des générateurs aléatoires [Paraskevopoulou et al., 2015]. M. Carlier, C. Dubois et A. Gotlieb ont formellement certifié en Coq un solveur de contraintes, qui constitue une partie d'un environnement de test [Carlier et al., 2012]. Un solveur de contraintes certifié sur un domaine fini de tableaux nécessite des générateurs séquentiels certifiés de ces structures pour explorer leur domaine. En complément du test aléatoire, nous considérons le test exhaustif borné (BET), présenté dans la partie 2.2.2.1 du chapitre 2. Nous abordons ici le BET avec des algorithmes générant tous les tableaux ayant une structure et une longueur données.

Ce chapitre est une contribution au BET de fonctions C dont une entrée est un tableau représentant une structure combinatoire. Par exemple, un tableau sans répétition de n entiers compris entre 0 et $n-1$ représente une permutation de ces n entiers. On dit alors que le tableau satisfait deux propriétés structurelles : la propriété d'avoir des valeurs comprises entre 0 et $n-1$, qui lui donne la structure d'endofonction sur $\{0,\ldots,n-1\}$, et la propriété de non-répétition, qui lui donne la structure d'injection. Conjointement, ces deux propriétés donnent au tableau la structure de permutation. Pour énumérer ces structures, les combinatoriciens proposent des algorithmes efficaces [Kreher et al., 1999] et des catalogues de programmes qui les implémentent, tels que le fxtbook [Arndt, 2010]. Nous montrons ici comment les combinatoriciens peuvent tirer profit des méthodes de vérification du génie logiciel pour concevoir et vérifier ces programmes. Nous étayons cette affirmation dans ce chapitre, qui présente des vérifications déductives automatiques d'algorithmes combinatoires, ce qu'aucun travail antérieur n'a abordé jusqu'à présent.

Nous portons notre attention sur les algorithmes d'énumération combinatoire qui génèrent séquentiellement toutes les structures de même taille, selon un ordre total prédéfini. Nous désignons ici par générateur séquentiel un tel algorithme, lorsqu'il est composé de deux fonctions : la première fonction construit la plus petite structure de taille donnée selon l'ordre prédéfini, et la seconde fonction modifie une structure quelconque pour construire la structure suivante de même taille selon cet ordre. L'efficacité d'un générateur séquentiel est liée à sa capacité à ne pas construire de structures intermédiaires qui ne satisfont pas les propriétés attendues. Nous présentons une approche uniforme pour la spécification et la preuve de ces générateurs séquentiels de tableaux structurés, implémentés par des fonctions C.

Nous considérons trois propriétés comportementales pour ces fonctions de génération. La propriété de correction affirme que les deux fonctions génèrent des tableaux satisfaisant les contraintes prescrites. La propriété de progression affirme que la deuxième fonction génère un tableau supérieur à son tableau d'entrée selon l'ordre lexicographique. Cette propriété implique la terminaison des appels répétés à la seconde fonction. La propriété d'exhaustivité affirme que le générateur n'omet aucune solution. Selon l'approche de preuve présentée dans la partie 2.2.3 du chapitre 2, nous vérifions statiquement les propriétés de correction et de progression. Dans ce but, nous spécifions formellement le comportement des fonctions de génération de tableaux. Nous les annotons ensuite avec des invariants et variants de boucle de manière à ce que leurs contrats formels soient automatiquement prouvés. Ces démonstrations (appelées preuves dans toute la suite), sont complétées par des tests, entre autres pour s'assurer de l'exhaustivité des générateurs séquentiels.

Pour accélérer la production de nouveaux générateurs séquentiels vérifiés, nous proposons plusieurs patrons de programmation et de spécification. Le premier patron généralise le principe d'énumération selon un ordre lexicographique par modification de la fin du tableau, adopté par de nombreux algorithmes efficaces. Le deuxième patron permet de spécifier et vérifier rapidement un générateur séquentiel qui procède par filtrage parmi des structures plus générales. Il est complété par un patron décrivant comment transformer uniformément en fonction booléenne toute contrainte sur un tableau exprimée en logique du premier ordre. Ces patrons mettent en lumière l'aspect générique de ces modes de génération et ont contribué à l'élaboration de deux générateurs génériques qui permettent une automatisation de notre démarche.

Les contributions de ce chapitre sont :

— le codage en C et la spécification en ACSL d'algorithmes de génération de tableaux structurés,
— la preuve formelle automatique des propriétés de correction et de progression de ces programmes vis-à-vis de leur spécification formelle,
— des patrons généraux d'aide à l'écriture de ces programmes spécifiés,
— deux générateurs séquentiels génériques de tableaux structurés (par révision du suffixe et par filtrage), et
— une bibliothèque `enum` de générateurs séquentiels vérifiés, disponible sous forme d'une archive `enum.*.tar.gz` à l'adresse http://members.femto-st.fr/richard-genestier/en.

Après avoir défini la notion de générateur séquentiel de tableaux structurés, la partie 3.2 présente la génération selon un ordre lexicographique, à travers un exemple fil-rouge et un patron général. La partie 3.3 illustre la génération par filtrage avec le même exemple et propose des patrons rendant cette technique facile à appliquer pour implémenter et vérifier de nombreux générateurs séquentiels. Les générateurs séquentiels produits à l'aide de ces patrons forment une bibliothèque présentée dans la partie 3.4. Des perspectives ouvertes par ce travail sont ensuite proposées dans la partie 3.5.

Les fichiers sources des générateurs présentés dans ce chapitre sont disponibles dans les répertoires `fxtbook/` et `generation/` de la bibliothèque `enum`.

3.2 Principes et exemple

Nous présentons ici les principes de génération et de vérification de tableaux structurés mis en œuvre dans ce chapitre. Nous définissons tout d'abord dans la partie 3.2.1 la notion de générateur séquentiel. Les parties 3.2.2 et 3.2.4 présentent le principe de génération dans l'ordre lexicographique, respectivement à partir d'un exemple fil-rouge et d'un patron formel, tandis que la partie 3.2.3 présente une formalisation de la propriété de progression.

3.2.1 Fonctions de génération

Dans tout ce qui suit, les valeurs des tableaux sont des entiers mathématiques. La génération exhaustive bornée de tableaux n'a de sens que s'il n'existe qu'un nombre fini de tableaux de chaque longueur. À cette fin, les valeurs des tableaux sont supposées être majorées et minorées par deux entiers C, dont la valeur absolue est généralement faible, de sorte que les nombres de tableaux à générer ne deviennent pas trop grands. En outre, il peut souvent être supposé que tous les calculs permettant d'obtenir les nouvelles valeurs d'un tableau sont effectués dans ces limites. Sous ces hypothèses, les valeurs des tableaux peuvent être représentées en toute sécurité par des entiers C de type **`int`**, sans risque de débordement arithmétique.

Nous étudions les algorithmes d'énumération qui génèrent séquentiellement des tableaux de même taille, selon un ordre lexicographique induit par un ordre sur les éléments des tableaux. Un tel ordre peut être défini comme suit :

Définition 13 : Ordre lexicographique

Soit A un ensemble muni d'un ordre strict $<$ (relation binaire irréflexive et transitive). On appelle ordre lexicographique (induit par $<$ sur les tableaux), la relation binaire, notée $\prec$, telle que, pour tout entier $n \geq 0$ et pour tous les tableaux b et c de taille n dont les éléments sont dans A, on a $b \prec c$ si et seulement s'il existe un indice i $(0 \leq i \leq n-1)$ tel que $b[i] < c[i]$ et $b[j] = c[j]$ pour tout j entre 0 et $i-1$ inclus.

La relation binaire $\prec$ ainsi définie est un ordre strict, qui est total si l'ordre sur A l'est. Dans toute la suite, A est l'ensemble des entiers relatifs (ou des entiers représentables de type `int` en C, puisqu'on ignore ici les problèmes de dépassement de capacité arithmétique), muni de son ordre strict total naturel, tel que $i < i+1$ pour tout entier i.

Considérons une famille `z` de tableaux C structurés de taille `n` dont les éléments sont de type `int`. Un générateur séquentiel d'une telle famille est constitué de deux fonctions C, appelées fonctions de génération (séquentielle).

— La première fonction

```
int first_z(int a[], int n, ...)
```

génère le premier tableau `a` de taille `n` de la famille `z`. Elle retourne `1` s'il existe au moins un tableau de cette taille. Sinon, cette fonction retourne `0`.

— La seconde fonction

```
int next_z(int a[], int n, ...)
```

(appelée fonction de succession), retourne `1` et génère dans le tableau `a` de taille `n` le prochain élément de la famille `z` qui suit immédiatement celui qui est stocké dans le tableau `a` à l'appel de la fonction, si ce tableau n'est pas le dernier de la famille `z`. Sinon, elle retourne `0`.

Dans le profil de ces deux fonctions C, les pointillés représentent d'autres paramètres éventuellement requis pour la génération du tableau structuré. On se limite ici aux cas où aucun de ces paramètres n'est une structure supplémentaire. D'autre part, les tableaux sont supposés alloués et de taille suffisante à l'appel de ces fonctions.

Un usage typique de ces deux fonctions pour générer successivement dans l'unique variable `a` tous les tableaux de longueur `n` de la famille `z` consiste en un appel à la première fonction `first_z(a,n,...);`, suivi d'un traitement du premier tableau, puis d'un traitement des tableaux suivants dans le corps d'une boucle `while (next_z(a,n,...) == 1)`.

Selon [Ruskey, 2003], "*For many combinatorial objects, the fastest known algorithms for listing [...] are with respect to lexicographic order.*". Nos connaissances en combinatoire énumérative ne nous permettent pas d'apprécier la pertinence de cette affirmation très générale. Nous avons temporairement choisi d'y adhérer, et nous avons implémenté divers algorithmes de génération dans un ordre lexicographique. Les fonctions `next_...` de ces programmes suivent toutes le même principe, appelé ici "révision du suffixe", que nous décrivons sur un exemple, avant de le généraliser en proposant un patron de code C et d'annotations ACSL pour cette fonction de génération.

3.2.2 Exemple

En guise de fil rouge, considérons l'exemple des fonctions à croissance limitée, qui servira à expliquer les formalismes utilisés et leur sémantique.

Définition 14 : Fonction à croissance limitée

Une fonction à croissance limitée (RGF, pour Restricted Growth Function) de taille n est une endofonction a de $\{0, \ldots, n-1\}$ telle que $a(0) = 0$ et $a(k) \le a(k-1)+1$ pour $1 \le k \le n-1$.

Cette définition est issue de [Arndt, 2010, page 325], où les RGF sont appelées RGS (pour "Restricted Growth String").

On peut représenter une endofonction a de $\{0, \ldots, n-1\}$, et donc une RGF, par le tableau C de ses images :

0	1	$\ldots$	$n-1$
$a(0)$	$a(1)$	$\ldots$	$a(n-1)$

Ce tableau est de taille n et ses valeurs sont des entiers de $\{0, \ldots, n-1\}$. Le fxtbook propose un algorithme efficace [Arndt, 2010, page 325] pour calculer la RGF qui suit immédiatement une RGF r donnée, dans l'ordre lexicographique croissant :

1. Trouver l'entier j maximal tel que $a(j) \le a(j-1)$.
2. Si un tel entier existe, incrémenter la valeur $a(j)$ et fixer $a(i) = 0$ pour tout $i > j$. Les autres valeurs de a restent inchangées.
3. Sinon, la génération est terminée, a était la plus grande RGF.

Par exemple, selon cet ordre, les cinq RGFs de taille 3 sont 000, 001, 010, 011 et 012 (dans cette notation, le tableau a est représenté par le mot $a[0]a[1]\cdots a[n-1]$ de ses valeurs). Toujours selon cet ordre, la première RGF est la fonction constante égale à 0.

La spécification ACSL (resp. le code C) des fonctions `first_rgf` et `next_rgf` est donné dans le listing 3.1 (resp. 3.2).

```
1  #include "global.h"
2  /*@ predicate is_non_neg(int *a, ℤ n) = ∀ ℤ i; 0 ≤ i < n⇒ a[i] ≥ 0;
3    @ predicate is_le_pred(int *a, ℤ n) = ∀ ℤ i; 1 ≤ i < n⇒ a[i] ≤ a[i-1]+1;
4    @ predicate is_rgf(int *a, ℤ n) = a[0] == 0 ∧ is_non_neg(a,n)
5    @  ∧ is_le_pred(a,n); */
6
7  /*@ requires n > 0 ∧ \valid(a+(0..n-1));
8    @ assigns a[0..n-1];
9    @ ensures is_rgf(a,n); */
10 void first_rgf(int a[], int n);
11
12 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
13   @ requires is_rgf(a,n);
14   @ assigns a[1..n-1];
15   @ ensures is_rgf(a,n);
16   @ ensures \result == 1⇒ lt_lex{Pre,Post}(a,n); */
17 int next_rgf(int a[], int n);
```

Listing 3.1 – Contrats des fonctions de génération des RGFs (fichier `rgf.h`).

Nous expliquons les aspects du langage de spécification ACSL que nous utilisons à travers l'exemple de la fonction `next_rgf`. Les annotations ACSL de la fonction `first_rgf` sont analogues.

Le fichier `rgf.h` donné dans le listing 3.1 et inclus dans le listing 3.2 est composé de trois prédicats et de deux contrats de fonction.

La propriété caractéristique des RGFs selon la définition 14 est exprimée en ACSL entre les lignes 2 et 5 du listing 3.1 par les trois prédicats `is_rgf`, `is_non_neg` et `is_le_pred`.

Avant l'entête d'une fonction, une annotation **`requires`** `R;` spécifie que la précondition `R` doit être vérifiée par les paramètres de la fonction lorsqu'elle est appelée. Ainsi, on exige dans la ligne 12 que le tableau `a` soit alloué en mémoire et de taille `n` strictement positive. Dans la ligne 13 on exige que le tableau `a` représente une RGF. Une annotation de la forme **`assigns`** `A;` déclare dans `A` les paramètres de la fonction qui peuvent être modifiés lors de son exécution. Ainsi, la ligne 14 déclare que tous les éléments du tableau `a` peuvent être modifiés, sauf le premier `a[0]`.

Une annotation **`ensures`** `E;` affirme que la postcondition `E` est vraie en sortie de fonction. La propriété de correction affirme que tous les tableaux générés satisfont la contrainte d'être des RGFs. Ainsi, dans la ligne 15, on affirme que le tableau `a` sera une RGF après exécution de la fonction. La postcondition de la ligne 16 est expliquée dans la partie 3.2.3.

```
1  #include "rgf.h"
2
3  /*@ predicate is_zero(int *a, ℤ b) = ∀ ℤ i; 0 ≤ i < b⇒ a[i] == 0; */
4
5  void first_rgf(int a[], int n) {
6   int k;
7   /*@ loop invariant 0 ≤ k ≤ n;
8     @ loop invariant is_zero(a,k);
9     @ loop assigns k, a[0..n-1];
10    @ loop variant n-k; */
11  for (k = 0; k < n; k++) a[k] = 0;
12 }
13
14 int next_rgf(int a[], int n) {
15  int rev,k;
16  /*@ loop invariant 0 ≤ rev ≤ n-1;
17    @ loop invariant (∀ ℤ j; rev < j < n⇒ a[j] > a[j-1]);
18    @ loop assigns rev;
19    @ loop variant rev; */
20  for (rev = n-1; rev ≥ 1; rev--) if (a[rev] ≤ a[rev-1]) break;
21  if (rev == 0) return 0;
22  a[rev]++;
23  /*@ loop invariant rev+1 ≤ k ≤ n;
24    @ loop invariant is_non_neg(a,k);
25    @ loop invariant is_le_pred(a,k);
26    @ loop assigns k, a[rev+1..n-1];
27    @ loop variant n-k; */
28  for (k = rev+1; k < n; k++) a[k] = 0;
29  return 1;
30 }
```

Listing 3.2 – Génération efficace de RGFs en C (fichier `rgf.c`).

Le fichier `rgf.c` montré dans le listing 3.2 est composé d'un prédicat et de deux définitions de fonctions : `first_rgf` et `next_rgf`. Le prédicat `is_zero` défini dans la ligne 3 est introduit pour exprimer l'invariant de boucle de la fonction `first_rgf` (ligne 8). Détaillons maintenant les instructions C du corps de la fonction `next_rgf`. Ligne 20, une boucle parcourt le tableau de droite à gauche pour trouver l'indice à partir duquel la fin du tableau sera modifiée. Cet indice est appelé indice de révision du tableau `a`. Ici l'indice de révision `rev` est atteint lorsqu'on rencontre l'élément du tableau d'indice maximal inférieur ou égal à son prédécesseur dans le tableau. Si la recherche échoue, alors la

structure finale est atteinte (ligne 21). Dans le cas contraire, le contenu du tableau est modifié à partir de l'indice de révision jusqu'à la fin, de sorte que le nouveau tableau satisfasse la contrainte et soit supérieur au tableau courant dans l'ordre lexicographique. La façon d'effectuer cette révision dépend de la contrainte que le tableau doit satisfaire. Pour les RGFs, la propriété `a[rev] <= a [rev-1]` de l'indice de révision `rev` permet d'incrémenter `a[rev]` (ligne 22) et de remplir le reste du tableau avec des `0` (ligne 28) afin d'obtenir le prochain tableau satisfaisant la contrainte de croissance limitée.

Dans le corps des fonctions, on utilise les annotations ACSL suivantes. Une annotation **`loop invariant`** `I;` immédiatement avant une boucle déclare que la formule `I` est un invariant (inductif) de cette boucle, c'est-à-dire une propriété qui doit être établie avant exécution de la boucle et préservée à chaque passage dans la boucle. Par exemple, avant la deuxième boucle de la fonction `next_rgf`, trois invariants de boucle affirment successivement que la variable de boucle `k` reste entre `rev+1` et `n` (ligne 23), que les `k` premiers éléments du tableau sont positifs ou nuls (ligne 24), et que la propriété `is_le_pred` est satisfaite jusqu'à `k` (ligne 25).

L'annotation **`loop assigns`** avant une boucle définit un sur-ensemble des variables dont la valeur peut être modifiée par une itération de la boucle. Par exemple

```
loop assigns x, t1[i], t2[a..b], t3[..];
```

signifie que les seules valeurs qui peuvent changer sont celles de `x`, de `t1[i]`, des éléments de `t2` entre les indices `a` et `b`, et de tous les éléments du tableau `t3`. En revanche cela ne signifie pas que toutes ces valeurs vont certainement être modifiées, et encore moins qu'elles seront modifiées à chaque itération. Ainsi on stipule dans la ligne 18 que seule la variable de boucle `rev` sera modifiée durant l'exécution de la boucle. Ligne 26 on spécifie que la variable de boucle `k` ainsi que les éléments du tableau d'indice supérieur ou égal à `rev+1` seront modifiés.

L'annotation **`loop variant`** `V;` définit un variant de boucle `V` qui peut être utilisé pour garantir la terminaison de la boucle. Cette expression entière doit rester positive ou nulle et décroître strictement entre deux itérations successives de la boucle. Par exemple, l'expression `n-k` est un variant de la boucle de la ligne 28 du listing 3.2. Les annotations ACSL de la fonction `first_rgf` sont similaires.

Supposons que le listing 3.2 soit le contenu d'un fichier `rgf.c`. La vérification statique de la fonction `next_rgf` avec Frama-C assisté de WP s'effectue par exécution de la commande `frama-c -wp-fct next_rgf rgf.c`. Frama-C indique si chaque condition de vérification générée par WP est prouvée par le solveur SMT Alt-Ergo, en indiquant la durée de chaque preuve. Les résultats de cette vérification sont détaillés dans la partie 3.4.

3.2.3 PROPRIÉTÉ DE PROGRESSION

La propriété de progression affirme que la fonction de succession génère les tableaux dans l'ordre lexicographique. Ceci est stipulé dans la postcondition ACSL

```
ensures \result == 1⇒ lt_lex{Pre,Post}(a,n);
```

ligne 16 du listing 3.1. L'annotation ACSL `lt_lex{L1,L2}(a,n)` formalise que le tableau `a` au label `L1` est lexicographiquement plus petit qu'au label `L2`. Un label représente une position dans le programme. Toute expression `e` en ACSL peut être écrite

\at(e,L), signifiant que e est évalué au label L. Les labels Pre, Here et Post sont prédéfinis et correspondent respectivement à l'état initial, courant et final de l'expression lors de l'exécution d'une fonction ou d'une boucle.

Le prédicat lt_lex et deux prédicats auxiliaires formalisent la définition 13 dans un fichier d'entête global.h inclus dans tous les générateurs. Ces définitions sont montrées dans le listing 3.3.

```
1 /*@ predicate is_eq{L1,L2}(int *a, ℤ i) =
2   @  ∀ ℤ j; 0 ≤ j < i⇒ \at(a[j],L1) == \at(a[j],L2);
3   @ predicate lt_lex_at{L1,L2}(int *a, ℤ i) =
4   @  is_eq{L1,L2}(a,i) ∧ \at(a[i],L1) < \at(a[i],L2);
5   @ predicate lt_lex{L1,L2}(int *a, ℤ n) =
6   @  ∃ ℤ i; 0 ≤ i < n ∧ lt_lex_at{L1,L2}(a,i); */
```

Listing 3.3 – Prédicats exprimant la propriété de progression (fichier global.h).

Le prédicat auxiliaire is_eq formalise le fait que le préfixe a[0..i-1] du tableau a est identique aux labels L1 et L2. Le second prédicat auxiliaire lt_lex_at stipule que le préfixe a[0..i-1] est identique aux labels L1 et L2, et que la valeur a[i] au label L1 est plus petite qu'au label L2.

3.2.4 PATRON DE GÉNÉRATION DANS L'ORDRE LEXICOGRAPHIQUE

La fonction next_rgf et la fonction de succession de nombreux autres générateurs séquentiels efficaces de tableaux structurés suivent un principe de conception ici appelé révision du suffixe. Le listing 3.4 présente ce principe comme un patron de conception (*design pattern*) composé d'un code C et d'annotations ACSL pour la fonction de succession next_z d' un générateur séquentiel dans l'ordre lexicographique.

La famille z de tableaux structurés est définie par une contrainte formalisée par le prédicat is_z montré ligne 4 du listing 3.4. La fonction de succession next_z révise le suffixe de son tableau d'entrée a en deux étapes. Tout d'abord, elle cherche une position à partir de laquelle la fin du tableau sera modifiée. Cet indice est l'indice de révision du tableau a. Il doit satisfaire une certaine condition de révision. En second lieu, la fonction modifie le contenu du tableau a à partir de l'indice de révision jusqu'à la fin du tableau. La condition de révision est formalisée par le prédicat is_rev (ligne 5) et la fonction booléenne b_rev (déclarée et spécifiée lignes 8-13). La boucle dans la ligne 32 parcourt le tableau d'entrée a de droite à gauche pour trouver l'indice de révision rev. L'invariant de boucle ligne 29 établit que l'index de révision est l'index maximal qui vérifie la condition de révision. Si la recherche échoue (ligne 33), le tableau en entrée est le dernier de la famille et la fonction retourne 0. Sinon, la fonction suffix révise le tableau a à partir de l'index de révision jusqu'à la fin, comme spécifié par la clause assigns ligne 17. La postcondition ligne 18 établit que la valeur a[rev] du tableau a est augmentée à l'index de révision rev. Dans ce patron, les prédicats is_z et is_rev ne sont pas définis. C'est la raison pour laquelle ils sont déclarés en ACSL dans un bloc axiomatique, de la ligne 3 à la ligne 6 du listing 3.4.

La fonction de succession suit le modèle de révision du suffixe si elle satisfait les propriétés de correction et de progression (respectivement spécifiées lignes 24 et 25), mais aussi la propriété qu'elle construit toujours le tableau suivant, c.a.d. qu'il n'existe aucun tableau dans la famille z entre les tableaux d'entrée et de sortie de cette fonction, pour l'ordre lexicographique strict et total. Un ingrédient de la spécification de cette propriété est l'invariant de boucle ligne 29. Sa vérification est en outre discutée dans la partie 3.4.2.

```
1 #include "global.h"
2
3 /*@ axiomatic preds {
4   @  predicate is_z(int *a, ℤ n) reads a[0..n-1];
5   @  predicate is_rev(int *a, ℤ n, ℤ i) reads a[0..n-1];
6   @ } */
7
8 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
9   @ requires 0 ≤ rev ≤ n-1;
10  @ assigns \nothing;
11  @ ensures \result == 0 ∨ \result == 1;
12  @ ensures \result⇔ is_rev(a,n,rev); */
13 int b_rev(int a[], int n, int rev);
14
15 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
16  @ requires 0 ≤ rev ≤ n-1;
17  @ assigns a[rev..n-1];
18  @ ensures a[rev] > \at(a[rev],Pre); */
19 void suffix(int a[], int n, int rev);
20
21 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
22  @ requires is_z(a,n);
23  @ assigns a[0..n-1];
24  @ ensures soundness: is_z(a,n);
25  @ ensures \result == 1⇒ lt_lex{Pre,Post}(a,n); */
26 int next_z(int a[], int n) {
27  int rev;
28  /*@ loop invariant -1 ≤ rev ≤ n-1;
29   @ loop invariant (∀ ℤ j; rev < j < n⇒ ! is_rev(a,n,j));
30   @ loop assigns rev;
31   @ loop variant rev; */
32  for (rev = n-1; rev ≥ 0; rev--) if (b_rev(a,n,rev)) break;
33  if (rev == -1) return 0;
34  suffix(a,n,rev);
35  return 1;
36 }
```

Listing 3.4 – Patron de fonction de succession par révision du suffixe.

Supposons que le listing 3.4 soit le contenu d'un fichier `suffix.c`. La postcondition exprimant la propriété de progression ainsi que l'invariant de boucle ligne 29 sont automatiquement prouvés par WP à l'aide de la commande

`frama-c -wp suffix.c -wp-prop=-soundness.`

On notera que l'invariant de boucle ligne 29 n'est pas nécessaire pour prouver la propriété de progression. En effet, l'algorithme mis en œuvre par la fonction `next_z` correspond à la définition de l'ordre lexicographique : il laisse un préfixe du tableau `a` inchangé et augmente sa valeur à l'indice de révision. La propriété de progression est donc génériquement vérifiée, car elle est une conséquence du patron. En revanche, la propriété de correction ne peut être vérifiée à ce niveau de généralité car elle dépend de la contrainte inhérente au tableau `a`.

En instanciant avec le code approprié les prédicats `is_z` et `is_rev` ainsi que les sous-fonctions `b_rev` et `suffix` de ce patron, on obtient un générateur d'une famille `z` de tableaux structurés dans l'ordre lexicographique, dont la propriété de progression peut être vérifiée automatiquement, en supposant que les sous-fonctions satisfont leurs contrats. Lors de cette instanciation du patron, les contrats des sous-fonctions doivent être complétés de sorte que la propriété de correction puisse également être vérifiée automatiquement. Pour les générateurs simples, il est plus facile de remplacer les appels aux sous-fonctions par une séquence d'instructions. Par exemple, nous pouvons obtenir la fonction de succession `next_rgf` du listing 3.2 en remplaçant les appels `b_rev(a,n,rev);` et

`suffix(a,n,rev);` respectivement par la déclaration

```
a[rev] <= a[rev-1];
```

et la séquence d'instructions

```
a[rev]++;
for (k = rev+1; k < n; k++) a[k] = 0;
```

3.3 Génération par filtrage

La génération par filtrage consiste à sélectionner parmi des structures celles qui satisfont une propriété donnée. Plus le filtrage élimine de structures, moins cet générateur séquentiel est efficace. Cependant, il peut être suffisant pour détecter une erreur dans un programme ou pour valider une conjecture portant sur ces structures. D'autre part, cette approche basique de génération fournit rapidement un premier générateur, dont l'implémentation, la spécification et la vérification déductive sont obtenues avec un coût minimal, comme nous le verrons dans cette partie.

La partie 3.3.1 illustre le principe de génération par filtrage sur l'exemple des RGFs. La partie 3.3.2 formalise ce principe, en proposant un patron commun pour tous les générateur séquentiels par filtrage. La propriété de correction des fonctions de génération de ce patron est automatiquement prouvée. Pour instancier ce patron, il est nécessaire d'implémenter la contrainte des sous-structures, sous la forme d'une fonction booléenne. La partie 3.3.3 propose un patron général pour cette fonction booléenne et sa spécification. La propriété de correction de la fonction booléenne par rapport à la contrainte est également automatiquement prouvée.

3.3.1 Exemple

D'après la définition 14, la famille des RGFs est une sous-famille de la famille des endofonctions de $\{0,\ldots,n-1\}$. Supposons que nous ayons déjà implémenté, spécifié et vérifié automatiquement un générateur séquentiel des endofonctions de $\{0,\ldots,n-1\}$, constitué des deux fonctions `first_endofct(a,n)` et `next_endofct(a,n)`. Nous utilisons ce générateur séquentiel pour implémenter très rapidement un générateur séquentiel des RGFs, par filtrage des endofonctions de $\{0,\ldots,n-1\}$ qui sont des RGFs.

Ce générateur est présenté dans le listing 3.5. Les fonctions de génération `first_rgf(a,n)` et `next_rgf(a,n)` appellent la fonction booléenne C `b_rgf`, qui caractérise une RGF parmi les endofonctions sur $\{0,\ldots,n-1\}$. Le prédicat ACSL `is_rgf` est ici la conjonction des prédicats `is_le_pred` et `is_endofct`. Les contrats des fonctions `first_rgf` et `next_rgf` ne sont pas montrés parce qu'ils sont similaires à ceux du listing 3.1, sauf la postcondition exprimant la propriété de correction de `next_rgf` qui est maintenant

```
ensures \result == 1⇒ is_rgf(a,n);
```

car le tableau `a` n'encode ici une RGF que si les fonctions `first_rgf` et `next_rgf` retournent la valeur `1`.

```
1 /*@ requires n > 0;
2   @ requires \valid(a+(0..n-1));
3   @ requires is_endofct(a,n);
4   @ assigns \nothing;
5   @ ensures \result == 0
6   @ ∨ \result == 1;
7   @ ensures \result == 1⇔ is_rgf(a,n);
8   @*/
9 int b_rgf(int a[], int n) {
10  int i;
11  if (a[0] ≠ 0) return 0;
12  /*@ loop invariant 1 ≤ i ≤ n;
13    @ loop invariant is_le_pred(a,i);
14    @ loop assigns i;
15    @ loop variant n-i; */
16  for (i = 1; i < n; i++)
17   if (a[i] > a[i-1]+1) return 0;
18  return 1;
19 }
20
21 int first_rgf(int a[], int n) {
22  int tmp;
23  tmp = first_endofct(a,n);
24  /*@ loop invariant
25    @  tmp ≠ 0⇒ is_endofct(a,n);
26    @ loop assigns a[0..n-1],tmp; */
27  while (tmp ≠ 0 ∧ b_rgf(a,n) == 0) {
28   tmp = next_endofct(a,n);
29  }
30  if (tmp == 0) { return 0; }
31  return 1;
32 }
33
34 int next_rgf(int a[], int n) {
35  int tmp = 0;
36  /*@ loop assigns a[0..n-1], tmp;
37    @ loop invariant is_endofct(a,n);
38    @ loop invariant
39    @  le_lex{Pre,Here}(a,n); */
40  do {
41   tmp = next_endofct(a,n);
42  } while (tmp ≠ 0 ∧ b_rgf(a,n) == 0);
43  if (tmp == 0) { return 0; }
44  return 1;
45 }
```

Listing 3.5 – Génération des RGFs par filtrage.

```
1 /*@ predicate le_lex{L1,L2}(int *a, ℤ n) = lt_lex{L1,L2}(a,n)
2   @ ∨ is_eq{L1,L2}(a,n); */
3
4 /*@ lemma trans_le_lt_lex{L1,L2,L3}: ∀ int *a; ∀ ℤ n;
5   @  (le_lex{L1,L2}(a,n) ∧ lt_lex{L2,L3}(a,n))⇒ lt_lex{L1,L3}(a,n); */
```

Listing 3.6 – Prédicat et lemme pour spécifier et prouver la progression d'une fonction de succession par filtrage.

Le prédicat et le lemme définis dans le listing 3.6 sont respectivement introduits pour spécifier un invariant (lignes 38-39) pour la boucle filtrante de la fonction de succession et pour prouver automatiquement cet invariant. La propriété de progression, assurée par la fonction de succession des endofonctions, est automatiquement prouvée pour la fonction de succession `next_rgf`. Le tableau courant est en effet identique au précédent au début de la boucle **do .. while** (lignes 40-42). Le lemme de pseudo-transitivité `trans_le_lt_lex` aide le prouveur à établir la propriété de progression – exprimée avec le prédicat `lt_lex` – à partir de cet invariant de boucle et du contrat de la fonction appelée `next_endofct`.

3.3.2 Patron général de génération par filtrage

La génération des RGFs par filtrage se généralise à de nombreuses familles de tableaux structurés, définies par ajout de propriétés à des structures plus générales. Le listing 3.7 propose un patron général pour la génération de tableaux d'une famille `z` en filtrant les tableaux `x` qui satisfont la contrainte supplémentaire `is_y` implémentée par la fonction booléenne `b_y`. Les tableaux de la famille `x` sont supposés vérifier la contrainte `is_x`. Dans ce patron, les prédicats ACSL `is_x`, `is_y` et `is_z` ne sont pas définis. C'est la raison pour laquelle ils sont déclarés en ACSL dans un bloc axiomatique, de la ligne 1 à la ligne 8 du listing 3.7.

```
1  /*@ axiomatic preds {
2    @  predicate is_x(int *a, ℤ n)
3    @   reads a[0..n-1];
4    @  predicate is_y(int *a, ℤ n)
5    @   reads a[0..n-1];
6    @  predicate is_z(int *a, ℤ n) =
7    @   is_x(a,n) ∧  is_y(a,n);
8    @ } */
9
10 /*@ requires n > 0 ∧  \valid(a+(0..n-1));
11   @ assigns a[0..n-1];
12   @ ensures
13   @  \result == 0 ∨  \result == 1;
14   @ ensures
15   @  \result == 1⇒  is_x(a,n); */
16 int first_x(int a[], int n);
17
18 /*@ requires n > 0 ∧  \valid(a+(0..n-1));
19   @ requires is_x(a,n);
20   @ assigns a[0..n-1];
21   @ ensures
22   @  \result == 0 ∨  \result == 1;
23   @ ensures
24   @  \result == 1⇒  is_x(a,n);
25   @ ensures \result == 1
26   @ ⇒  lt_lex{Pre,Post}(a,n); */
27 int next_x(int a[], int n);
28
29 /*@ requires n > 0;
30   @ requires \valid(a+(0..n-1));
31   @ assigns \nothing;
32   @ ensures \result == 0
33   @  ∨  \result == 1;
34   @ ensures \result == 1
35   @ ⇔  is_y(a,n); */
36 int b_y(int a[], int n);
37
38 /*@ requires n > 0 ∧  \valid(a+(0..n-1));
39   @ assigns a[0..n-1];
40   @ ensures
41   @  \result == 0 ∨  \result == 1;
42   @ ensures
43   @  \result == 1⇒  is_z(a,n); */
44 int first_z(int a[], int n) {
45  int tmp;
46  tmp = first_x(a,n);
47  /*@ loop invariant tmp ≠ 0
48    @ ⇒  is_x(a,n);
49    @ loop assigns a[0..n-1], tmp; */
50  while (tmp ≠ 0 ∧  b_y(a,n) == 0) {
51   tmp = next_x(a,n);
52  }
53  if (tmp == 0) { return 0; }
54  return 1;
55 }
56
57 /*@ requires n > 0 ∧  \valid(a+(0..n-1));
58   @ requires is_z(a,n);
59   @ assigns a[0..n-1];
60   @ ensures
61   @  \result == 0 ∨  \result == 1;
62   @ ensures \result == 1⇒  is_z(a,n);
63   @ ensures \result == 1
64   @ ⇒  lt_lex{Pre,Post}(a,n); */
65 int next_z(int a[], int n) {
66  int tmp;
67  /*@ loop invariant is_x(a,n);
68    @ loop assigns a[0..n-1], tmp;
69    @ loop invariant
70    @   le_lex{Pre,Here}(a,n); */
71  do {
72   tmp = next_x(a,n);
73  } while (tmp ≠ 0 ∧  b_y(a,n) == 0);
74  if (tmp == 0) { return 0; }
75  return 1;
76 }
```

Listing 3.7 – Patron de génération par filtrage.

En supposant que les fonctions `b_y`, `first_x` et `next_x` satisfont leurs spécifications, la correction des fonctions `first_z` et `next_z` est prouvée automatiquement avec Frama-C et WP assistés d'Alt-Ergo.

Le générateur séquentiel des RGFs par filtrage (listing 3.5) s'obtient en instanciant le patron général comme suit : on remplace respectivement `x`, `y` et `z` par `endofct`, `rgf` et `rgf`, et on implémente la propriété `is_rgf` sous forme d'une fonction booléenne. D'autres exemples de générateurs séquentiels utilisant ce patron sont donnés dans la partie 3.4

Ainsi, à partir d'un générateur séquentiel d'une famille de tableaux structurés spécifié et vérifié, on obtient rapidement des générateurs séquentiels spécifiés de leurs sous-familles, le processus pouvant s'itérer. Leur vérification automatique dépend essentiellement de celle de la fonction booléenne qui implémente leur contrainte.

3.3.3 Patron général des fonctions booléennes

Nous proposons à présent un patron pour le contrat ACSL et le code C d'une fonction booléenne correspondant à une propriété structurelle d'un tableau `t` exprimée en logique du premier ordre avec égalité, à partir du terme `t[i]` qui désigne le `i`-ème élément du

tableau t. Si la propriété n'est qu'une combinaison booléenne de prédicats atomiques, la correspondance est évidente : les opérateurs booléens (comme la conjonction ou la négation) soit existent en C, soit peuvent être facilement exprimés par une combinaison d'opérateurs C. Ainsi, les cas intéressants sont les formules avec quantificateurs. Le cas d'un seul quantificateur est trop restrictif. Le cas général, avec des quantificateurs imbriqués et combinés avec des opérateurs booléens de manière arbitraire, serait trop lourd à formaliser. Le résultat serait peu lisible. Nous avons choisi de présenter le cas de deux quantificateurs imbriqués. Il suffit pour donner une idée de ce que serait le cas général, et il est utile en lui-même.

Les listings 3.8 et 3.9 proposent un patron, nommé ALLEX, qui couvre tous les cas où la propriété est de la forme $\forall i. 0 \leq i < n \Rightarrow (\exists j. 0 \leq j < n \wedge \varphi)$, où φ est une formule sans quantificateurs exprimant une propriété dépendant de i et de j et portant sur un tableau de taille n. Cette propriété est décomposée en trois prédicats is_x1(a,n), is_x2(a,n,v1) et is_x3(a,n,v1,v2), correspondant respectivement à la propriété universelle complète, à la sous-formule existentielle et à la propriété φ qu'elle quantifie, pour un tableau a de taille n. Le paramètre supplémentaire v1 du prédicat is_x2 correspond à la variable libre i des sous-formules $(\exists j. 0 \leq j < n \wedge \varphi)$ et φ. De même, le paramètre supplémentaire v2 du prédicat is_x3 correspond à la variable libre j de φ. Cette formule sans quantificateurs φ étant quelconque, le prédicat correspondant is_x3 n'est pas défini, mais seulement déclaré dans un bloc axiomatique, dans les lignes 1 à 3 du listing 3.8.

```
1 /*@ axiomatic preds {
2   @  predicate is_x3(int *a, ℤ n, ℤ v1, ℤ v2) reads a[0..n-1];
3   @ }
4   @ predicate is_x2_gen(int *a, ℤ n, ℤ v1, ℤ v2) =
5   @  ∃ ℤ i2; 0 ≤ i2 < v2 ∧  is_x3(a,n,v1,i2);
6   @ predicate is_x2(int *a, ℤ n, ℤ v1) = is_x2_gen(a,n,v1,n);
7   @ predicate is_x1_gen(int *a, ℤ n, ℤ v1) = ∀ ℤ i1; 0 ≤ i1 < v1⇒ is_x2(a,n,i1);
8   @ predicate is_x1(int *a, ℤ n) = is_x1_gen(a,n,n); */
```

Listing 3.8 – Prédicats d'une propriété de la forme ∀∃.

```
1 /*@ requires n ≥ 0;
2   @ requires \valid(a+(0..n-1));
3   @ assigns \nothing;
4   @ ensures \result == 0 ∨
5   @  \result == 1;
6   @ ensures \result == 1⇔
7   @  is_x3(a,n,v1,v2); */
8 int b_x3(int a[],int n,int v1,int v2);
9
10 /*@ requires n ≥ 0;
11   @ requires \valid(a+(0..n-1));
12   @ assigns \nothing;
13   @ ensures \result == 0 ∨
14   @  \result == 1;
15   @ ensures \result == 1⇔
16   @  is_x2(a,n,v1); */
17 int b_x2(int a[],int n,int v1) {
18  int i;
19  /*@ loop invariant 0 ≤ i ≤ n;
20    @ loop invariant
21    @  ! is_x2_gen(a,n,v1,i);
22    @ loop assigns i;
23    @ loop variant n-i; */
24  for (i = 0; i < n; i++)
25   if (b_x3(a,n,v1,i) == 1) return 1;
26  return 0;
27 }
28
29 /*@ requires n ≥ 0 ∧  \valid(a+(0..n-1));
30   @ assigns \nothing;
31   @ ensures \result == 0 ∨
32   @  \result == 1;
33   @ ensures \result == 1⇔
34   @  is_x1(a,n); */
35 int b_x1(int a[],int n) {
36  int i;
37  /*@ loop invariant 0 ≤ i ≤ n;
38    @ loop invariant is_x1_gen(a,n,i);
39    @ loop assigns i;
40    @ loop variant n-i; */
41  for (i = 0; i < n; i++)
42   if (b_x2(a,n,i) == 0) return 0;
43  return 1;
44 }
```

Listing 3.9 – Patron d'une fonction boolénne associée à une propriété de la forme ∀∃.

On dit que la fonction booléenne b_x1 implémente le prédicat is_x1 pour exprimer

qu'elle retourne `1` si ses paramètres satisfont le prédicat, et `0` sinon. De même, les fonctions booléennes `b_x2` et `b_x3` implémentent respectivement les prédicats `is_x2` et `is_x3`. Pour k =`1`,`2`, la fonction `is_x`k implémente une boucle qui évalue successivement le prédicat `is_x`$(k+1)$ pour tous les éléments du tableau. Puisque la fonction booléenne `b_x1` implémente une propriété universelle, elle retourne `0` dès qu'un élément qui ne satisfait pas la formule quantifiée est rencontré. Sinon, elle retourne `1`. L'invariant de boucle spécifie que la propriété `is_x2` est vraie jusqu'à l'indice courant `i` de parcours du tableau. Dualement, puisque la fonction booléenne `b_x2` implémente une propriété existentielle, elle retourne `1` dès qu'un élément satisfaisant la formule quantifiée est rencontré. Sinon, elle retourne `0`. L'invariant de boucle spécifie que la propriété `is_x2` est fausse jusqu'à l'indice courant `i` de parcours du tableau. Ces invariants de boucle sont spécifiés à l'aide d'une généralisation du prédicat `is_x`k, nommée `is_x`k`_gen`, définie dans les lignes 4 et 5 du listing 3.8.

Supposons que les listings 3.8 et 3.9 forment dans cet ordre le contenu d'un fichier `allex.c`. Admettons que la fonction booléenne `b_x3` est conforme à sa spécification. Par la commande

```
frama-c -wp allex.c -wp-skip-fct b_x3
```

on prouve alors automatiquement que les autres fonctions satisfont leur spécification.

Une application immédiate du patron précédent est la génération des endosurjections par filtrage des endofonctions, ce qui constitue un moyen de générer les permutations. On appelle endosurjection de $\{0,\dots,n-1\}$ toute endofonction de $\{0,\dots,n-1\}$ qui est une surjection. Une endosurjection f satisfait la propriété $\forall i.\, 0 \le i < n \Rightarrow (\exists j.\, 0 \le j < n \wedge f(j) = i)$. Un générateur séquentiel des endosurjections s'obtient simplement en complétant les listings 3.8 et 3.9 avec le listing 3.7 du patron de génération par filtrage, en renommant `x1`, `x2`, `x`, `y` et `z` respectivement en `im`, `eq_im`, `endofct`, `im` et `surj`, en définissant le prédicat `is_x3` par

```
predicate is_x3(int *a, ℤ n, ℤ v1, ℤ v2) = a[v2] == v1;
```

et en implémentant la fonction `b_x3` avec l'instruction unique

```
return(a[v2] == v1);
```

A partir du générateur séquentiel d'endofonctions déjà utilisé pour les RGFs dans la partie 3.3.1, la mise au point et la vérification automatique de ce générateur séquentiel d'endosurjections s'effectue en quelques minutes. Après ce travail minimal, il est possible d'apporter diverses simplifications à ce générateur séquentiel, tout en préservant sa vérification automatique. Par exemple, on peut supprimer le premier paramètre du prédicat `is_x3`, qui n'est pas utilisé dans cet exemple.

Pour d'autres cas d'alternance de deux quantificateurs, tels que les propriétés de la forme $\exists i.\, 0 \le i < n \wedge (\forall j.\, 0 \le j < n \Rightarrow \varphi)$ et $\forall i.\, 0 \le i < n \wedge (\forall j.\, 0 \le j < n \Rightarrow \varphi)$, où φ est une propriété sans quantificateurs portant sur un tableau de taille n, nous proposons aussi des patrons, respectivement nommés EXALL et ALL2.

3.4 Bibliothèque de générateurs formellement vérifiés

Les patrons présentés dans les parties précédentes ont été implémentés et instanciés pour produire une bibliothèque `enum` de générateurs séquentiels vérifiés de tableaux

Famille de tableaux	C	ACSL	nb. VC	durée (s)
SUFFIX	9	12	26	4.176
FILTERING	14	33	51	8.285
ALLEX	11	28	40	3.050
EXALL	12	27	40	2.925
ALL2	40	28	40	2.950
FCT	13	25	42	8.821
SUBSET	13	22	40	8.178
ENDOFCT	4	12	17	3.587
RGF ⊂ ENDOFCT	25	27	69	10.365
SORTED1 ⊂ FCT	19	27	67	9.360
SORTED2 ⊂ FCT	28	48	103	12.122
INJ ⊂ FCT	29	42	91	12.392
SURJ ⊂ FCT	29	40	103	12.980
COMB ⊂ FCT	21	28	67	10.011
PERM ⊂ INJ (ENDOINJ)	4	11	15	3.130
PERM ⊂ SURJ (ENDOSURJ)	4	11	15	2.722
PERM = ENDOFCT ∧ INJ	17	21	60	8.660
PERM = ENDOFCT ∧ SURJ	28	40	102	12.260
DERANG ⊂ PERM	20	27	66	10.040
INVOL ⊂ PERM	20	27	66	9.901
FFI ⊂ INVOL	20	27	64	11.071
CPERM ⊂ PERM	27	44	97	12.537
CINVOL ⊂ INVOL	27	58	96	14.112
CFFI ⊂ CINVOL	20	28	66	10.942
RGF	13	28	41	10.295
SORTED	13	30	44	34.489
COMB	18	33	46	Timeout
PERM	23	29	52	10.583
INVOL	23	29	41	13.043

TABLE 3.1 – Résultats de vérification des générateurs de la bibliothèque `enum`.

structurés. Afin d'assurer l'existence d'un nombre fini de tableaux de chaque longueur, tous les générateurs de la bibliothèque raffinent un générateur de tableaux dont le codomaine est fini. Ce générateur, nommé FCT, génère toutes les fonctions de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k-1\}$ dans l'ordre lexicographique croissant.

La littérature en combinatoire énumérative [Arndt, 2010, Ruskey, 2003] fournit de nombreux algorithmes permettant la génération des stuctures combinatoires classiques, telles que les n tuples, les permutations ou les combinaisons de k éléments parmi n. En utilisant les patrons de la partie 3.3, nous obtenons rapidement des générateurs de ces structures par filtrage parmi les fonctions. Ensuite, nous implémentons, spécifions et vérifions des générateurs plus efficaces issus de la littérature, en instanciant le patron de révision du suffixe de la partie 3.2.4. Le résultat de ce travail est détaillé dans la partie 3.4.1. La preuve ou le test de propriétés supplémentaires de ces générateurs sont ensuite détaillés dans la partie 3.4.2. Enfin, deux générateurs génériques correspondants à chaque mode de génération sont présentés dans la partie 3.4.3.

Les structures énumérées sont représentées par un tableau de taille strictement positive

n, dont les éléments sont des entiers entre 0 et k. Cet entier k est égal à $n-1$ si la structure ne nécessite qu'un seul paramètre. Ces tableaux sont toujours générés dans l'ordre lexicographique induit par la relation de précédence stricte $<$ sur les entiers (définition 13).

3.4.1 GÉNÉRATEURS PROUVÉS

Les résultats expérimentaux sont collectés dans la table 3.1. La première colonne donne le nom de la famille de structures générées. Ces noms sont introduits dans la suite de cette partie. Le nombre de lignes de code, comptabilisé dans la deuxième colonne, ne prend pas en compte les lignes constituées uniquement d'accolades. Le nombre de lignes d'ACSL est donné dans la troisième colonne. La quatrième colonne indique le nombre de conditions de vérification générées par WP (incluant celle du lemme `trans_le_lt_lex`, présenté dans le listing 3.6, dans le cas d'une génération par filtrage). La cinquième colonne donne la durée de la preuve de ces buts par les solveurs Alt-Ergo, CVC3 et CVC4, en secondes (seul le meilleur temps de preuve de chaque but parmi les trois solveurs est pris en compte). Hormis pour FCT et SUBSET, une extension à deux minutes de la durée de calcul par défaut du greffon WP est nécessaire.

Le premier bloc de lignes de la table 3.1 concerne les patrons de révision du suffixe et de filtrage présentés dans les parties 3.2.4 et 3.3.2. Les patrons ALLEX, EXALL et ALL2 correspondent respectivement aux contraintes du premier ordre de la forme ∀∃, ∃∀ et ∀∀. Seule la propriété de progression est prouvée pour le patron SUFFIX.

Le deuxième bloc concerne le générateur FCT gènérant toutes les fonctions de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k-1\}$.

Le troisième bloc de la table 3.1 concerne les spécialisations, notion que nous définissons maintenant. Lorsqu'une famille de tableaux structurés a d'autres paramètres que sa taille, on peut fixer la valeur de certains de ces paramètres et obtenir ainsi facilement d'autres générateurs séquentiels. On dit qu'on a spécialisé la famille. Par exemple, la spécialisation de la famille des fonctions de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k\}$ pour $k = 1$ donne les fonctions booléennes, qui codent la famille SUBSET des sous-ensembles d'un ensemble à n éléments [Arndt, 2010, page 203]). La spécialisation avec $k = n-1$ donne la famille ENDOFCT des endofonctions de $\{0, \ldots, n-1\}$.

Le quatrième bloc de la table 3.1 concerne la génération par filtrage (partie 3.3). Nous notons z ⊂ x un générateur de structures z qui opère par filtrage parmi les structures plus générales x. Par exemple, RGF ⊂ ENDOFCT désigne un générateur séquentiel des fonctions à croissance limitée par filtrage parmi les endofonctions (présenté dans la partie 3.3.1). A partir du générateur séquentiel FCT des fonctions de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k\}$, nous avons généré par filtrage :

- Les tableaux triés de longueur n dont les éléments appartiennent à $\{0, \ldots, k-1\}$, en comparant chaque valeur du tableau à la suivante si elle existe (SORTED1),
- les tableaux triés de longueur n dont les éléments appartiennent à $\{0, \ldots, k-1\}$, en comparant chaque valeur du tableau aux autres, par instanciation du patron ALL2 (SORTED2),
- les injections de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k\}$ $(n \leq k+1)$, générateur séquentiel INJ ⊂ FCT,
- les surjections de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k\}$ $(n \geq k+1)$, générateur séquentiel SURJ ⊂ FCT, et

— les combinaisons de p éléments choisis parmi n (générateur séquentiel COMB ⊂ FCT). La combinaison $\{e_0, \ldots, e_{p-1}\}$, avec $0 \le e_0 < \ldots < e_{p-1} \le n-1$ est représentée par la fonction c de $\{0, \ldots, p-1\}$ dans $\{0, \ldots, n-1\}$ telle que $c[i] = e_i$. La famille COMB correspond donc à la famille des fonctions strictement croissantes de $\{0, \ldots, p-1\}$ dans $\{0, \ldots, n-1\}$.

Par spécialisation ou filtrage, on peut produire quatre générateurs séquentiels des permutations de $\{0, \ldots, n-1\}$ (générateur PERM) :

— PERM ⊂ INJ (aussi nommé ENDOINJ) par spécialisation des injections de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k-1\}$ (pour $k = n$)
— PERM ⊂ SURJ (aussi nommé ENDOSURJ) par spécialisation des surjections de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k-1\}$ (pour $k = n$)
— PERM = ENDOFCT ∧ INJ (aussi nommé INJENDO) par filtrage des injections parmi les endofonctions sur $\{0, \ldots, n-1\}$
— PERM = ENDOFCT ∧ SURJ (aussi nommé SURJENDO) par filtrage des surjections parmi les endofonctions sur $\{0, \ldots, n-1\}$.

Nous notons ici z = x ∧ y un générateur des structures z qui sélectionne parmi les structures plus générales x celles qui vérifient la propriété supplémentaire des structures y. Le générateur PERM = ENDOFCT ∧ SURJ est détaillé dans la partie 3.3.3. Les permutations ainsi obtenues permettent d'obtenir d'autres familles par filtrages successifs, par exemple les dérangements (permutations sans point fixe) de $\{0, \ldots, n-1\}$ (DERANG ⊂ PERM), les involutions de $\{0, \ldots, n-1\}$ (INVOL ⊂ PERM), et les involutions sans point fixe de $\{0, \ldots, 2n-1\}$ (FFI ⊂ INVOL). A partir de la famille des permutations (resp. involutions et involutions sans point fixe), nous pouvons à nouveau obtenir par filtrage des permutations (resp. involutions et involutions sans point fixe) connectées (CPERM ⊂ PERM, CINVOL ⊂ INVOL et CFFI ⊂ CINVOL). La notion de permutation connectée, définie dans la partie 2.1.3.2 du chapitre 2, sera utilisée dans le chapitre 10.

Famille de tableaux	nb. VC	durée (s)
FCT	43	6.858
SUBSET	41	7.277
RGF	42	8.760
SORTED	45	8.007
COMB	48	29.094
PERM	51	9.595

TABLE 3.2 – Résultats de vérification pour les algorithmes efficaces avec une assertion finale.

Le cinquième bloc de la table 3.1 concerne les générateurs efficaces implémentés par l'instanciation du patron de génération par révision du suffixe présenté dans la partie 3.2. Le générateur RGF génère les fonctions à croissance limitée grâce à l'algorithme donné dans [Arndt, 2010, page 325], comme détaillé dans la partie 3.2.2. Le générateur SORTED produit tous les tableaux triés de $\{0, \ldots, n-1\}$ dans $\{0, \ldots, k-1\}$ par révision du suffixe, plus efficacement que les générateurs SORTED1 et SORTED2 qui procèdent par filtrage. Le générateur COMB produit les combinaisons de p éléments parmi n en utilisant l'algorithme [Arndt, 2010, page 178]. Le générateur PERM produit les permutations sur $\{0, \ldots, n-1\}$ par une adaptation de l'algorithme [Arndt, 2010, page 243].

En complément, le sixième bloc de la table 3.1 présente un générateur INVOL qui produit les involutions sur $\{0, \ldots, n-1\}$ par une adaptation de l'algorithme [Arndt, 2010, page 284-285], plus efficacement que le générateur INVOL ⊂ PERM. Contrairement aux autres

générateurs efficaces présentés ici, il ne génère pas les involutions dans l'ordre lexicographique. Il sera utilisé dans le chapitre 5.

La colonne 4 de la table 3.1 montre que les preuves de ces générateurs optimisés sont plus complexes. Ils nécessitent une plus longue durée de vérification que les générateurs obtenus par filtrage. En particulier, la propriété de progression du générateur COMB est difficile à prouver, comme l'indique le timeout, renvoyé par les prouveurs, indiqué dans la table 3.1. Ce timeout signifie que les prouveurs ne parviennent pas à décharger la condition de vérification correspondant à la propriété de progression du générateur optimisé de combinaisons, malgré l'extension de la durée de calcul à deux minutes par condition de vérification.

Cependant, l'assertion supplémentaire

```
/*@ assert lt_lex_at{Pre,Here}(a,rev); */
```

à la fin de la fonction de succession réduit sensiblement la durée des preuves les plus longues, comme montré dans la table 3.2. Cette assertion spécifie en effet que le tableau courant (évalué au label **Here**) est supérieur au précédent (évalué au label **Pre**) à partir de l'index de révision `rev`. Cette assertion aide le prouveur à choisir l'index `rev` pour instancier le quantificateur existentiel du prédicat `lt_lex`. En particulier, cette assertion élimine le timeout pour le générateur COMB.

3.4.2 Autres propriétés

Afin d'augmenter encore davantage la confiance dans la correction des générateurs séquentiels proposés, nous avons validé deux propriétés supplémentaires, détaillées dans les parties 3.4.2.1 et 3.4.2.2.

3.4.2.1 Non modification

Nous avons également prouvé la postcondition

```
ensures \result == 0⇒ is_eq{Pre,Post}(a,n);
```

pour la fonction de succession `next_z` du générateur par révision du suffixe. Elle exprime le fait que le tableau `a` n'est pas modifié quand la fonction retourne 0, c.a.d. lorsqu'aucun indice de révision n'a été trouvé, indiquant que le tableau maximal dans l'ordre lexicographique a été atteint.

Cette propriété n'est pas vérifiée par la génération par filtrage qui instancie le patron présenté dans la partie 3.3.2, quand le tableau maximal de la sous-famille `z`, disons `m_z`, n'est pas le tableau maximal de la famille `x`, disons `m_x`. Dans ce cas la fonction `next_z` considère tous les tableaux plus grands que `m_z` dans la famille `x` tant que le contenu du tableau a changé, jusqu'à atteindre `m_x` et retourner `0`.

3.4.2.2 Exhaustivité

Il existe plusieurs façons de vérifier la propriété d'exhaustivité affirmant que tous les tableaux caractérisés par une certaine contrainte sont générés :

(i) On peut stocker tous les tableaux générés dans un tableau global puis spécifier et prouver qu'il contient tous les tableaux répondant à la contrainte. L'exhaustivité a été formalisée de cette manière pour un générateur produisant toutes les solutions du problème des n reines [Filliâtre, 2012]. La preuve formelle de l'exhaustivité des solutions utilise l'outil de vérification Why3 [Bobot et al., 2013] et nécessite des étapes interactives. Nous rejetons cette solution parce que nous souhaitons rester dans le cadre d'une approche où la vérification est complètement automatisée.
(ii) Une autre solution est de préciser que la fonction de succession calcule toujours le tableau suivant, c.a.d. qu'il n'existe aucun tableau ayant la contrainte voulue entre les tableaux d'entrée et de sortie, pour l'ordre lexicographique strict et total. Cette quantification sur les tableaux rend cette propriété plus difficile à prouver automatiquement que les propriétés de correction et de progression.
(iii) Lorsque les propriétés de correction et de progression sont déjà prouvées, la propriété d'exhaustivité peut être validée jusqu'à une certaine longueur de tableau en comptant simplement le nombre de tableaux générés et en le comparant au nombre attendu, obtenu soit dans une séquence de l'encyclopédie des séquences de nombres entiers (OEIS [The OEIS Foundation Inc., 2010], voir la partie 2.1.2 du chapitre 2), soit à l'aide de formules de dénombrement connues, implémentées sous forme de fonctions C.

Notre travail suit cette troisième voie. Cette validation a été effectuée pour toutes les structures de la bibliothèque, en augmentant la longueur des tableaux générés jusqu'à la limite du plus grand entier positif représentable permettant de générer des structures en un temps raisonnable.

Nous avons également effectué des validations relatives d'un générateur séquentiel par rapport à un autre. Cette validation est souvent facile à mettre en œuvre, d'une part parce que nous obtenons rapidement un générateur séquentiel de référence par filtrage, d'autre part parce que ce générateur séquentiel et le générateur séquentiel efficace comparé produisent les structures dans le même ordre lexicographique. Dans ce cas, le stockage des structures énumérées est inutile : il suffit de générer en parallèle les structures avec chaque générateur séquentiel et de tester leur égalité. Pour cette validation, les générateurs par filtrage RGF $\subset$ FCT, COMB $\subset$ FCT, SORTED $\subset$ FCT et PERM $\subset$ FCT ont été utilisés en tant qu'implémentation de référence pour valider les générateurs optimisés RGF, COMB, SORTED et PERM.

3.4.3 Générateurs génériques

Tous les générateurs séquentiels de la bibliothèque `enum` sont des instanciations des patrons présentés dans les parties 3.2.4 et 3.3.2. A partir de cette remarque, nous avons mis au point un générateur séquentiel générique pour chacun des patrons.

Ces générateurs sont montrés dans le listing 3.10. La fonction de succession `next_suffix` du générateur générique par révision du suffixe, présentée de la ligne 1 à la ligne 11, prend en paramètre, en plus du tableau `a` et de sa taille `n`, la fonction booléenne de condition de révision `b_rev` ainsi que la fonction de révision du suffixe `suffix`. Si ces fonctions sont définies et implémentées pour une famille de tableaux `z`, un exemple d'utilisation de ce générateur est un appel

```
first_z(a,n,...);
```

à la première fonction de génération, qui produit le premier tableau, puis un traitement

des tableaux suivants dans le corps d'une boucle

```
while(next_suffix(a,n,(& b_rev),(& suffix)) == 1).
```

```
1 int next_suffix(int a[], int n,
2  int (*b_rev)(int a[], int n, int rev),
3  int (*suffix)(int a[], int n, int rev)) {
4  int rev;
5
6  for (rev = n-1; rev ≥ 0; rev--)
7   if ((*b_rev)(a,n,rev)) break;
8  if (rev == -1) return 0;
9  suffix(a,n,rev);
10 return 1;
11 }
12
13 int first_filter(int a[], int n,
14  int (*first_x)(int a[], int n),
15  int (*next_x)(int a[], int n),
16  int (*filter) (int a[], int n)) {
17  int tmp = (*first_x)(a,n);
18
19  while (tmp ≠ 0 ∧ (*filter)(a,n) == 0)
20   tmp = (*next_x)(a,n);
21  if (tmp == 0) return 0;
22  return 1;
23 }
24
25 int next_filter(int a[], int n,
26  int (*next_x)(int a[], int n),
27  int (*filter) (int a[], int n)) {
28  int tmp;
29
30  do tmp = (*next_x)(a,n);
31  while (tmp ≠ 0 ∧ (*filter)(a,n) == 0);
32  if (tmp == 0) return 0;
33  return 1;
34 }
```

Listing 3.10 – Générateurs génériques.

Le deuxième générateur générique, correspondant au patron de génération par filtrage présenté dans le listing 3.7, est constitué de deux fonctions. La fonction `first_filter`, implémentée lignes 13-23 du listing 3.10, génère le premier tableau d'une famille `z` par filtrage parmi une famille plus vaste `x`. Elle requiert en paramètre, en plus du tableau `a` et de sa taille `n`, les deux fonctions de génération `first_x` et `next_x` des tableaux de la famille `x`, ainsi que la fonction `filter` filtrant les éléments de la famille `x`. La fonction `next_filter`, implémentée quant à elle lignes 25-34 du listing 3.10, génère le tableau suivant de la famille `z`, et ne requiert pas la fonction `first_x` en paramètre. Par exemple, la génération de tous les tableaux de taille `n` de la famille RGF par filtrage parmi la famille `endofct` des endofonctions sur $\{0, \ldots, n-1\}$ consiste en un appel

```
first_filter(a,n,(& first_endofct),(& next_endofct),(& b_rgf));
```

qui produit le premier tableau, suivi par un traitement des tableaux suivants dans le corps d'une boucle

```
while(next_filter(a,n,(& next_endofct),(& b_rgf)) == 1).
```

Ces générateurs génériques ne comportent pas d'annotations ACSL car ce langage de spécification ne permet pas encore d'annoter des fonctions passées en paramètre d'une autre fonction. Par exemple, dans le contrat de la fonction `first_filter`, il faudrait pouvoir spécifier le contrat de `(*first_x)`. Ce manque rejoint la problématique évoquée dans [Tushkanova et al., 2010] concernant la spécification modulaire de classes et méthodes Java génériques.

L'utilisateur de la bibliothèque doit donc choisir la méthode adaptée au but recherché : Si l'on désire obtenir un générateur séquentiel formellement vérifié, il faut instancier les patrons présentés dans la partie 3.3.2. Si l'on préfère obtenir rapidement un générateur en privilégiant l'aspect test, un générateur générique est tout indiqué. Des utilisations de générateurs génériques pour générer des cartes étiquetées seront détaillées dans le chapitre 5.

3.5 Conclusion et perspectives

Notre bibliothèque `enum` trouve son utilité dans le domaine du test exhaustif borné, mais il existe d'autres outils et techniques permettant de renforcer la confiance dans les programmes manipulant des données structurées, présentés dans la partie 2.2.2.2 du chapitre 2. Par rapport à ces différents outils, notre approche est très spécifique. En effet, nous fournissons un générateur dédié pour chaque structure étudiée. L'utilisateur doit écrire les fonctions de génération de données ainsi que les prédicats, ce qui nous rapproche sur ce point de l'outil UDITA [Gligoric et al., 2010]. Bien que cette approche nécessite davantage de travail, elle garantit néanmoins de trouver les plus petits contre-exemples. Elle est donc complémentaire aux autres approches. En outre, nous fournissons des générateurs formellement vérifiés qui peuvent constituer des parties d'outils de vérification certifiés. À notre connaissance, la vérification déductive de générateurs exhaustifs de structures de données avec contraintes n'avait jamais encore été abordée.

L'énumération de tableaux structurés par taille croissante jusqu'à une taille donnée peut s'avérer très utile pour tester automatiquement des programmes dont ces structures sont des entrées. Elle permet en outre au combinatoricien de valider expérimentalement des conjectures sur les structures qu'il étudie. Les algorithmes d'énumération efficace posent aussi d'intéressants problèmes de vérification déductive. Pour toutes ces raisons, nous avons entrepris de développer une bibliothèque `enum` de programmes d'énumération formellement spécifiés et vérifiés automatiquement. Afin de réduire le coût de leur spécification et de leur vérification déductive, nous proposons des patrons généraux pour différentes familles de générateurs, dont l'instanciation donne facilement des programmes corrects. En particulier, un patron pour le principe de base de génération par filtrage permet d'implémenter de nombreux générateurs à partir d'un nombre très restreint de générateurs classiques. Les propriétés de correction et de progression de ces générateurs sont automatiquement vérifiées. Nous fournissons également un patron pour des générateurs plus efficaces, dont la propriété de progression est automatiquement vérifiée. La correction des générateurs obtenus par instanciation de ce patron est plus difficile à vérifier, mais la plate-forme Frama-C et ses greffons apportent une aide importante.

Les deux modes de génération détaillés dans ce chapitre, la spécification de la propriété de correction des algorithmes étudiés, ainsi qu'une partie de la bibliothèque, ont été présentés lors de la conférence JFLA 2015 [Genestier et al., 2015a]. La spécification de la propriété de progression des algorithmes étudiés, les patrons de génération formellement vérifiés, ainsi qu'une bibliothèque enrichie, ont été présentés lors de la conférence TAP 2015 [Genestier et al., 2015b].

La bibliothèque `enum` de générateurs formellement vérifiés issue de ce travail, ainsi que les générateurs séquentiels génériques, constituent un outil que chacun peut utiliser comme aide à la génération de structures combinatoires. Dans notre cas, c'est un outil essentiel dans notre objectif de génération de structures plus complexes - telles que les cartes enracinées - encodées en C, comme nous allons le découvrir dans certains des chapitres suivants.

4

Opérations sur les permutations

Les définitions des (hyper)cartes combinatoires du chapitre 2 sont basées sur la notion de permutation. Nous proposons donc dans ce chapitre une formalisation des permutations et d'opérations sur les permutations qui seront utilisées dans la suite de ce mémoire, pour construire des (hyper)cartes.

Il existe diverses bibliothèques de formalisation des permutations avec preuves interactives, présentées dans la partie 2.3.1 du chapitre 2. L'objectif de notre travail étant l'automatisation des preuves formelles, nous n'avons pas utilisé ces bibliothèques, et préféré une formalisation des opérations sur les permutations en C et de leurs propriétés en ACSL, pour une vérification par preuve avec le greffon WP de Frama-C.

La partie 4.1 définit quatre opérations sur les permutations, puis décrit leur implémentation en C. Leur spécification en ACSL et leur preuve formelle sont détaillées dans la partie 4.2. L'une de ces opérations permet la génération exhaustive des permutations. La partie 4.3 détaille la vérification par preuve de cette affirmation. Enfin, la partie 4.4 donne les perspectives de cette étude.

Ce chapitre étend la communication [Genestier et al., 2016]. Les fichiers sources de cette étude sont disponibles dans le répertoire `perm/` de la bibliothèque `enum`.

4.1 Opérations sur les permutations

Nous présentons respectivement dans les parties 4.1.1, 4.1.2, 4.1.3 et 4.1.4 les opérations d'insertion, de retrait, de somme directe et de composition sur les permutations, ainsi que leurs implémentations par des fonctions C.

4.1.1 Insertion dans une permutation

Dans une étude sur les permutations ayant un nombre fixé de cycles [Baril, 2007], J.-L. Baril a défini une opération qui correspond à l'insertion de la valeur n **après** une valeur i dans un cycle d'une permutation p de taille n. Nous avons identifié dans le cadre de nos travaux sur les cartes détaillés dans le chapitre 8 le besoin d'une variante de cette opération qui insert la valeur n **avant** une valeur dans un cycle d'une permutation. Nous définissons ici cette opération.

Définition 15 : Insertion dans une permutation

Soit p une permutation sur $\{0,...,n-1\}$ et i un entier naturel entre 0 et n inclus. On appelle insertion avant i dans p, et on désigne par $insert$, l'opération qui ajoute à p le cycle (n) si $i = n$ et qui insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i \leq n-1$.

La permutation qui résulte de cette opération est désignée par $insert(p,i)$. C'est une permutation de taille $n+1$.

Exemple 11. *À partir de la permutation $p = (0\ 1\ 3)\ (2\ 4) = 1\ 3\ 4\ 0\ 2 \in S_5$, on peut définir 6 permutations de S_6 par application de l'opération $insert$ avec $i \in \{0,...,5\}$. Par exemple, $insert(p,0)$ est la permutation $(0\ 1\ 3\ 5)\ (2\ 4) = 1\ 3\ 4\ 5\ 2\ 0$, $insert(p,4)$ est la permutation $(0\ 1\ 3)\ (2\ 5\ 4) = 1\ 3\ 5\ 0\ 2\ 4$ et $insert(p,5)$ est la permutation $(0\ 1\ 3)\ (2\ 4)\ (5) = 1\ 3\ 4\ 0\ 2\ 5$ qui ajoute le point fixe 5 à p.*

Nous présentons dans le listing 4.1 deux implémentations de cette opération par une fonction C : l'une qui étend la permutation p en place, et l'autre qui duplique cette permutation. Ces deux fonctions utilisent la représentation fonctionnelle de la permutation p présentée dans la partie 2.1.3.

```
1  void insert_inplace(int p[], int i, int n) {
2   if (0 ≤ i ∧ i ≤ n) {
3    p[n] = i;
4    for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
5   }
6  }
7
8  void insert(int p[], int i, int q[], int n) {
9   if (0 ≤ i ∧ i ≤ n) {
10   q[n] = i;
11   for (int j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
12  }
13 }
```

Listing 4.1 – Fonctions d'insertion en C.

La première fonction `insert_inplace` prend en paramètres une permutation p, sa taille n et un entier i. Si ces paramètres permettent d'insérer la valeur n dans la permutation p, c'est-à-dire si $0 \leq i \leq n$, le comportement de la fonction est différent selon que i vaut n ou pas. Si $i = n$, la valeur n insérée crée un point fixe dans p (ligne 3). Dans ce cas la boucle de la ligne 4 n'a aucun effet sur p. Si $i < n$, la valeur n est insérée avant la valeur i (ligne 3) et après la valeur $p^{-1}(i)$ dans un cycle de p, par la boucle de la ligne 4.

La seconde fonction `insert` présentée lignes 8-13 effectue la même opération mais en construisant une nouvelle permutation q de taille $n+1$, ce qui nécessite une recopie des valeurs de la permutation p dans q (boucle ligne 11).

Une application directe de l'opération $insert$ sera détaillée dans la partie 4.3. Ces deux fonctions sont également des constituants des fonctions de construction de cartes qui seront implémentées dans le chapitre 9.

4.1.2 Retrait du maximum

Après avoir étudié l'ajout de la valeur n dans une permutation de taille n, nous considérons à présent son retrait.

Définition 16 : Retrait du maximum dans une permutation

Soit p une permutation de taille $n > 0$. On appelle retrait du maximum, et on désigne par $remove_max$, l'opération qui retire la valeur maximale $n-1$ de son cycle de p.

La permutation qui résulte de cette opération est désignée par $remove_max(p)$. C'est une permutation de taille $n-1$.

Exemple 12. *Considérons la permutation* $p = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4) \in S_5$. *On a alors* $remove_max(p, 5) = 1\ 3\ 2\ 0 = (0\ 1\ 3)\ (2)$.

Nous présentons dans le listing 4.2 deux implémentations de cette opération par une fonction C : l'une qui réduit la permutation p de taille $n > 0$ en place pour obtenir une permutation de taille $n-1$, et l'autre qui construit une nouvelle permutation q de taille $n-1$.

```
1 void remove_max_inplace(int p[], int n) {
2  for (int i = 0; i < n-1; i++) if (p[i] == n-1) p[i] = p[n-1];
3 }
4
5 void remove_max(int p[], int n, int q[]) {
6  for (int i = 0; i < n-1; i++) q[i] = (p[i] == n-1) ? p[n-1] : p[i];
7 }
```

Listing 4.2 – Fonctions de retrait du maximum en C.

La première fonction `remove_max_inplace` prend en paramètres une permutation p et sa taille n. La valeur $n-1$ est retirée dans un cycle de p, ce qui revient à placer $p[n-1]$ à la place $n-1$ dans la représentation fonctionnelle de p. La nouvelle permutation p est alors de taille $n-1$. Cette fonction sera utilisée dans la partie 4.3.

La seconde fonction `remove_max` effectue la même opération mais en construisant une nouvelle permutation q de taille $n-1$, ce qui nécessite une recopie des valeurs de la permutation p dans q (boucle ligne 6). Cette fonction sera utilisée dans le chapitre 9.

4.1.3 Somme directe de deux permutations

L'opération de somme directe de deux permutations est bien connue en combinatoire [Kitaev, 2011]. Cette opération permet de fusionner deux permutations par décalage des valeurs de l'une d'entre elles. Soit p une permutation sur $\{0, \dots, n-1\}$ et k un entier naturel. On appelle décalage de p selon k la permutation p' sur $\{k, \dots, n+k-1\}$ telle que $p'(i+k) = p(i) + k$. Informellement, l'opération $\oplus$ de somme directe fusionne deux permutations $p_1 \in S_{n_1}$ et $p_2 \in S_{n_2}$ afin d'obtenir une permutation $p = p_1 \oplus p_2 \in S_{n_1+n_2}$ dont la restriction sur $\{0, \dots, n_1 - 1\}$ est égale à p_1 et la restriction sur $\{n_1, \dots, n_1 + n_2 - 1\}$ est égale au décalage de p_2 selon n_1.

Définition 17 : Somme directe de deux permutations

La somme directe $p_1 \oplus p_2$ de deux permutations $p_1 \in S_{n_1}$ et $p_2 \in S_{n_2}$ est définie par
$(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$ et
$(p_1 \oplus p_2)(i) = p_2(i - n_1) + n_1$ pour $n_1 \leq i < n_1 + n_2$.

Exemple [13]**.** *Considérons les deux permutations* $p_1 = 2\ 1\ 0 = (0\ 2)\ (1) \in S_3$ *et* $p_2 = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4) \in S_5$*, ainsi que la permutation vide notée* $\emptyset$*. Alors* $p_1 \oplus \emptyset = \emptyset \oplus p_1 = p_1$*, le décalage de* p_2 *selon* 3 *est la permutation* $4\ 6\ 7\ 3\ 5 = (3\ 4\ 6)\ (5\ 7)$ *sur* $\{3, \ldots, 7\}$*, et* $p_1 \oplus p_2 = 2\ 1\ 0\ 4\ 6\ 7\ 3\ 5 = (0\ 2)\ (1)\ (3\ 4\ 6)\ (5\ 7)$.

Deux implémentations de la somme directe sous forme de deux fonctions C `sum` et `sum_inplace` sont présentées dans le listing 4.3. La fonction `sum` prend en paramètres une permutation p_1, sa taille n_1, une permutation p_2, sa taille n_2, et construit une nouvelle permutation $p = p_1 \oplus p_2$, résultat de la somme directe de p_1 et p_2. Grâce à une seule boucle, cette fonction recopie les valeurs de p_1 dans p (partie `p1[i]` des affectations de la boucle), effectue le décalage de la permutation p_2 selon n_1, et le recopie dans p à la suite des valeurs de p_1 (partie `p2[i-n1]+n1` des affectations de la boucle). Cette opération est l'un des constituants des opérations de construction de cartes définies dans le chapitre 9.

```
1 void sum(int p1[], int p2[], int p[], int n1, int n2) {
2  for (int i = 0; i < n1+n2; i++) p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
3 }
4
5 void sum_inplace(int p1[], int p2[], int n1, int n2) {
6  for (int i = n1; i < n1+n2; i++) p1[i] = p2[i-n1]+n1;
7 }
```

Listing 4.3 – Fonctions de somme directe de deux permutations en C.

La seconde fonction `sum_inplace` présentée lignes 5-7 effectue la même opération, mais la construction de la nouvelle permutation $p = p_1 \oplus p_2$ écrase la permutation p_1. Le décalage de la permutation p_2 selon n_1 est ici recopié dans p_1 à la suite de ses valeurs. Cette fonction sera utilisée dans le chapitre 10.

4.1.4 Composition

L'opération de composition (ou produit) de deux permutations p_1 et p_2 de taille n est définie dans la définition 2 du chapitre 2. Elle est implémentée dans le cadre de ce mémoire car elle intervient dans la construction de la permutation des faces d'une carte combinatoire.

```
1 void composition(int p1[], int p2[], int p[], int n) {
2  for (int i = 0; i < n; i++) p[i] = p2[p1[i]];
3 }
```

Listing 4.4 – Fonction de composition de deux permutations en C.

Une implémentation de la composition de deux permutations p_1 et p_2 sous forme d'une fonction C `composition` est présentée dans le listing 4.4. Cette fonction prend en paramètres une permutation p_1, une permutation p_2, leur taille n, et construit une nouvelle permutation $p = p_1\ p_2$, résultat de la composition de p_1 et p_2. Cette fonction sera utilisée dans les chapitres 9 et 10.

4.2 Spécification et preuve

Nous pouvons doter les fonctions présentées dans la partie 4.1 de contrats et d'annotations, afin de prouver automatiquement qu'elles préservent les permutations. Il existe

différentes manières de caractériser une permutation. Par exemple, bien qu'une permutation soit définie comme une bijection sur un ensemble fini, il est suffisant de la définir comme une endofonction injective ou surjective, puisque sur un ensemble fini l'injectivité et la surjectivité s'impliquent mutuellement. Nous avons choisi de définir une permutation comme une endofonction injective sur $\{0, \ldots, n-1\}$. Ainsi, nous spécifions une permutation à l'aide du prédicat ACSL `is_perm` présenté dans le listing 4.5 ligne 4. La propriété caractéristique d'une fonction est formalisée par le prédicat `is_fct` (déjà présenté dans le chapitre 3). La propriété d'injectivité est formalisée par le prédicat `is_linear`.

```
1 /*@ predicate is_fct(int *a, ℤ b, ℤ c) = ∀ ℤ i; 0 ≤ i < b⇒ 0 ≤ a[i] < c;
2   @ predicate is_linear(int *a, ℤ n) =
3   @  ∀ ℤ j; 0 ≤ j < n⇒ ∀ ℤ k; 0 ≤ k < n⇒ (j ≠ k⇒ a[j] ≠ a[k]);
4   @ predicate is_perm(int *a, ℤ n) = is_fct(a,n,n) ∧ is_linear(a,n); */
```

Listing 4.5 – Prédicats ACSL caractérisant une permutation.

Nous détaillons dans la partie 4.2.1 la spécification et la vérification par preuve des fonctions ne modifiant pas les tableaux en entrée, et dans la partie 4.2.2 la spécification et la vérification par preuve des fonctions modifiant l'un des tableaux en entrée.

4.2.1 Cas des fonctions ne modifiant pas les tableaux en entrée

Nous abordons tout d'abord la vérification par preuve des fonctions `insert`, `remove_max`, `sum` et `composition`. Les prédicats `is_loop_insert`, `is_remove`, `is_sum` et `is_comp`, présentés dans le listing 4.6, paraphrasent respectivement les instructions de boucle des fonctions `insert`, `remove_max`, `sum` et `composition`. Le corps de ces trois dernières fonctions n'étant composé que d'une boucle, les prédicats `is_remove`, `is_sum` et `is_comp` spécifient complètement le comportement des fonctions `remove_max`, `sum` et `composition`. La spécification complète de la fonction `insert` est donnée par le prédicat `is_insert` présenté lignes 3-4.

```
1 /*@ predicate is_loop_insert(int *q, int *p, ℤ b, ℤ c, ℤ d) =
2   @  ∀ ℤ i; 0 ≤ i < b⇒ q[i] == ((p[i] == c) ? d : p[i]);
3   @ predicate is_insert(int *q, int *p, ℤ b, ℤ c) =
4   @  0 ≤ b ≤ c⇒ (q[c] == b ∧ is_loop_insert(q,p,c,b,c));
5   @ predicate is_remove(int *q, int *p, ℤ b, ℤ c) =
6   @  ∀ ℤ i; 0 ≤ i < b⇒ q[i] == ((p[i] == c) ? p[c] : p[i]);
7   @ predicate is_sum(int *p, int *p1, int *p2, ℤ b, ℤ c) =
8   @  ∀ ℤ i; 0 ≤ i < c⇒ p[i] == ((i < b) ? p1[i] : p2[i-b]+b);
9   @ predicate is_comp(int *p, int *p1, int *p2, ℤ b) =
10  @  ∀ ℤ i; 0 ≤ i < b⇒ p[i] == p2[p1[i]]; */
```

Listing 4.6 – Prédicats ACSL pour les opérations sur les permutations.

Le listing 4.7 présente le contrat (lignes 1-5) et les annotations de boucle (lignes 9-13) de la fonction `insert`. Cette fonction manipulant plusieurs tableaux, il est nécessaire de garantir que ces tableaux sont alloués dans des zones distinctes de la mémoire, à l'aide du prédicat ACSL `separated` (ligne 2). L'invariant ligne 10 spécifie le comportement de la boucle grâce au prédicat `is_loop_insert`, afin d'aider les solveurs SMT à prouver les deux postconditions lignes 4-5. De plus, pour prouver la postcondition `ensures is_perm(q,n+1);`, les invariants de boucle de la ligne 11 spécifient qu'à chaque itération de boucle, les propriétés caractéristiques d'une permutation sont satisfaites jusqu'à l'indice de boucle.

```
1 /*@ requires 0 ≤ i ≤ n ∧ \valid(p+(0..n-1)) ∧ \valid(q+(0..n));
2   @ requires \separated(q+(0..n),p+(0..n-1)) ∧ is_perm(p,n);
3   @ assigns q[0..n];
```

```
4   @ ensures is_perm(q,n+1);
5   @ ensures is_insert(q,p,i,n); */
6 void insert(int p[], int i, int q[], int n) {
7  if (0 ≤ i ∧  i ≤ n) {
8   q[n] = i;
9   /*@ loop invariant 0 ≤ j ≤ n;
10    @ loop invariant is_loop_insert(q,p,j,i,n);
11    @ loop invariant is_fct(q,j,n+1) ∧ is_linear(q,j);
12    @ loop assigns j, q[0..n-1];
13    @ loop variant n-j; */
14   for(int j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
15  }
16 }
```

Listing 4.7 – Fonction `insert` annotée.

Remarquons que cette dernière postcondition est en réalité une conséquence de la post-condition `ensures is_insert(q,p,i,n);` caractérisant complètement le comportement de la fonction `insert`. Cette affirmation est prouvée formellement par le lemme ACSL suivant :

```
1 /*@ lemma insert2perm: ∀ int *p,*q; ∀ ℤ n,i;
2   @  (0 ≤ i ≤ n ∧  \valid(p+(0..n-1)) ∧  \valid(q+(0..n)) ∧
3   @  \separated(q+(0..n),p+(0..n-1)) ∧  is_perm(p,n) ∧  is_insert(q,p,i,n))
4   @ ⇒ is_perm(q,n+1); */
```

qui établit que si l'on applique l'opération *insert* à une permutation p de longueur n avec en paramètre l'entier i ($0 \leq i \leq n$), alors on obtient une permutation q de longueur $n+1$.

Les spécifications des fonctions `remove_max`, `sum` et `composition` suivent le même principe. Elles sont présentées respectivement dans les listings 4.8, 4.9 et 4.10. De même, la preuve de la préservation des permutations par ces fonctions est une conséquence de la preuve de leur caractérisation complète, établie respectivement par les postconditions

```
ensures is_remove(q,p,n-1,n-1);
```

```
ensures is_sum(p,p1,p2,n1+n2,n1);
```

et

```
ensures is_comp(p,p1,p2,n);
```

présentes dans les contrats de ces fonctions.

```
1 /*@ requires n > 0 ∧  \valid(p+(0..n-1)) ∧  \valid(q+(0..n-2));
2   @ requires \separated(q+(0..n-2),p+(0..n-1)) ∧  is_perm(p,n);
3   @ assigns q[0..n-2];
4   @ ensures is_perm(q,n-1);
5   @ ensures is_remove(q,p,n-1,n-1); */
6 void remove_max(int p[], int n, int q[]) {
7  int i;
8  /*@ loop invariant 0 ≤ i ≤ n-1;
9    @ loop invariant is_remove(q,p,i,n-1);
10   @ loop invariant is_fct(q,i,n-1) ∧  is_linear(q,i);
11   @ loop assigns i, q[0..n-2];
12   @ loop variant n-1-i; */
13  for(i = 0; i < n-1; i++) q[i] = (p[i] == n-1) ? p[n-1] : p[i];
14 }
```

Listing 4.8 – Fonction `remove_max` annotée.

```
1 /*@ requires n1 ≥ 0 ∧  n2 ≥ 0 ∧  \separated(p1+(0..n1-1),p2+(0..n2-1),p+(0..n1+n2-1));
2   @ requires \valid(p1+(0..n1-1)) ∧  \valid(p2+(0..n2-1)) ∧  \valid(p+(0..n1+n2-1));
3   @ requires is_perm(p1,n1) ∧  is_perm(p2,n2);
```

```
4   @ assigns p[0..n1+n2-1];
5   @ ensures is_perm(p,n1+n2);
6   @ ensures is_sum(p,p1,p2,n1+n2,n1); */
7 void sum(int p1[], int p2[], int p[], int n1, int n2) {
8  /*@ loop invariant 0 ≤ i ≤ n1+n2;
9    @ loop invariant is_sum(p,p1,p2,i,n1);
10   @ loop invariant is_fct(p,i,n1+n2) ∧ is_linear(p,i);
11   @ loop assigns i, p[0..n1+n2-1];
12   @ loop variant n1+n2-i; */
13  for (int i = 0; i < n1+n2; i++) p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
14 }
```

Listing 4.9 – Fonction `sum` annotée.

```
1 /*@ requires n > 0 ∧ \valid(p1+(0..n-1)) ∧ \valid(p2+(0..n-1)) ∧ \valid(p+(0..n-1));
2   @ requires \separated(p1+(0..n-1),p2+(0..n-1),p+(0..n-1));
3   @ requires is_perm(p1,n) ∧ is_perm(p2,n);
4   @ assigns p[0..n-1];
5   @ ensures is_perm(p,n);
6   @ ensures is_comp(p,p1,p2,n); */
7 void composition(int p1[], int p2[], int p[], int n) {
8  int i;
9  /*@ loop invariant 0 ≤ i ≤ n;
10   @ loop invariant is_comp(p,p1,p2,i);
11   @ loop invariant is_fct(p,i,n) ∧ is_linear(p,i);
12   @ loop assigns i, p[0..n-1];
13   @ loop variant n-i; */
14  for (i = 0; i < n; i++) p[i] = p2[p1[i]];
15 }
```

Listing 4.10 – Fonction `composition` annotée.

Les contrats de ces fonctions sont prouvés automatiquement, grâce aux invariants de boucle. Ces invariants étant des généralisations “naturelles” des postconditions qu'on veut démontrer, les preuves de ces fonctions ne présentent pas de difficulté importante. La situation est différente pour les versions `inplace` de ces fonctions étudiées dans la partie suivante.

4.2.2 Cas des fonctions modifiant un tableau en entrée

Les fonctions `insert_inplace`, `remove_max_inplace` et `sum_inplace` modifient en place le tableau en entrée. Pour leur spécification, nous adaptons les prédicats présentés dans le listing 4.6. En effet, ces prédicats doivent relier les valeurs d'un même tableau dans deux états différents de l'exécution du programme. Ceci est possible grâce aux prédicats hybrides paramétrés par des labels, comme décrit dans le chapitre 3. Rappelons que chaque expression `e` en ACSL peut être écrite `\at(e,L)`, signifiant que l'expression `e` est évaluée dans l'état identifié par le label `L`, et que les labels `Pre`, `Here` et `Post` correspondent respectivement à l'état initial, courant et final de la fonction ou de la boucle annotée.

Le listing 4.11 montre la spécification complète de la fonction `insert_inplace`. Son contrat est plus simple que celui de la fonction `insert`, puisqu'il n'y a pas de tableau `q`. Les annotations de boucle sont similaires à celles de la fonction `insert`, sauf l'invariant

```
loop invariant is_loop_insert(q,p,j,i,n);
```

qui est remplacé par deux invariants (lignes 8-9) qui relient les valeurs du tableau dans l'état courant (label `Here`) et dans l'état initial (label `Pre`) de la boucle. Le listing 4.12 définit les prédicats utilisés dans ces invariants. Le prédicat `is_loop_insert_inplace`, présenté lignes 1-3, paraphrase le code de la boucle. L'invariant de boucle

```
loop invariant is_loop_insert_inplace{Pre,Here}(p,j,i,n);
```

l'utilise pour décrire l'évolution de la partie $p[0..j-1]$ du tableau p, entre son indice 0 inclus et l'indice courant j de la boucle exclu. Le prédicat `is_insert_inplace`, présenté lignes 4-5 du listing 4.12, utilise le prédicat `is_loop_insert_inplace` pour spécifier totalement le comportement de la fonction `insert_inplace`. La postcondition ligne 4 du listing 4.11, établissant le comportement complet de la fonction `insert_inplace`, l'utilise en faisant référence aux labels `Pre` et `Post` qui relient les valeurs du tableau respectivement dans l'état initial et final de la fonction. Le prédicat `is_eq_gt`, présenté lignes 11-12 du listing 4.12, spécifie qu'un suffixe d'un tableau n'est pas modifié entre les états étiquetés par `L1` et `L2`. L'invariant de boucle

```
loop invariant is_eq_gt{Pre,Here}(p,j,n);
```

l'utilise pour spécifier que, lors de l'exécution de la boucle, la partie $p[j..n-1]$ du tableau p, de l'indice courant de boucle (inclus) à la fin du tableau, n'est pas modifiée. Cet invariant est ici nécessaire, car la clause `assigns p[0..n-1];` de la spécification stipule que toutes les valeurs du tableau p peuvent être modifiées par la boucle. Or, seule une valeur de ce tableau, non connue à l'avance, est en réalité modifiée. Comme il n'est pas possible de raffiner la clause `assigns`, nous compensons cette faiblesse de spécification par un invariant supplémentaire.

```
1  /*@ requires 0 ≤ i ≤ n ∧ \valid(p+(0..n-1)) ∧ is_perm(p,n);
2    @ assigns p[0..n];
3    @ ensures is_perm(p,n+1);
4    @ ensures is_insert_inplace{Pre,Post}(p,i,n); */
5  void insert_inplace(int p[], int i, int n) {
6   if (0 ≤ i ∧ i ≤ n) {
7    p[n] = i;
8    /*@ loop invariant 0 ≤ j ≤ n;
9      @ loop invariant is_loop_insert_inplace{Pre,Here}(p,j,i,n);
10     @ loop invariant is_eq_gt{Pre,Here}(p,j,n);
11     @ loop invariant is_fct(p,j,n+1) ∧ is_linear(p,j);
12     @ loop assigns j, p[0..n-1];
13     @ loop variant n-j; */
14   for(int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
15  }
16 }
```

Listing 4.11 – Fonction `insert_inplace` annotée.

```
1  /*@ predicate is_loop_insert_inplace{L1,L2}(int *p, ℤ b, ℤ c, ℤ d) =
2    @  ∀ ℤ j; 0 ≤ j < \at(b,L2)⇒
3    @   \at(p[j],L2) == ((\at(p[j],L1) == c ) ? d : \at(p[j],L1));
4    @ predicate is_insert_inplace{L1,L2}(int *p, ℤ b, ℤ c) =
5    @  0 ≤ b ≤ c⇒ (\at(p[c],L2) == b ∧ is_loop_insert_inplace{L1,L2}(p,c,b,c));
6    @ predicate is_remove_inplace{L1,L2}(int *p, ℤ b, ℤ c) =
7    @  ∀ ℤ j; 0 ≤ j < \at(b,L2)⇒
8    @   \at(p[j],L2) == ((\at(p[j],L1) == c ) ? \at(p[c],L1) : \at(p[j],L1));
9    @ predicate is_sum_inplace{L1,L2}(int *p1, int *p2, ℤ b, ℤ c) =
10   @  ∀ ℤ j; b ≤ j < \at(c,L2)⇒ \at(p1[j],L2) == \at(p2[j-b]+b,L1);
11   @ predicate is_eq_gt{L1,L2}(int *p, ℤ b, ℤ c) =
12   @  ∀ ℤ j; b ≤ j < c⇒ \at(p[j],L2) == \at(p[j],L1); */
```

Listing 4.12 – Prédicats ACSL pour les opérations modifiant leur tableau d'entrée.

Grâce à ces invariants, la correction de la fonction `insert_inplace` est prouvée automatiquement par WP. La difficulté ici était de penser à spécifier que la boucle ne modifie pas la partie $p[j..n-1]$ du tableau p.

```
1  /*@ requires n > 0 ∧ \valid(p+(0..n-1)) ∧ is_perm(p,n);
2    @ assigns p[0..n-2];
```

```
3   @ ensures is_perm(p,n-1);
4   @ ensures is_remove_inplace{Pre,Post}(p,n-1,n-1); */
5 void remove_max_inplace(int p[], int n) {
6  int i;
7  /*@ loop invariant 0 ≤ i ≤ n-1;
8    @ loop invariant is_remove_inplace{Pre,Here}(p,i,n-1);
9    @ loop invariant is_eq_gt{Pre,Here}(p,i,n-1);
10   @ loop invariant is_fct(p,i,n-1) ∧ is_linear(p,i);
11   @ loop assigns i, p[0..n-2];
12   @ loop variant n-1-i; */
13 for(i = 0; i < n-1; i++) if (p[i] == n-1) p[i] = p[n-1];
14 }
```

Listing 4.13 – Fonction `remove_max_inplace` annotée.

```
1 /*@ requires n1 ≥ 0 ∧ n2 ≥ 0 ∧ \valid(p2+(0..n2-1)) ∧ \valid(p1+(0..n1+n2-1));
2   @ requires \separated(p1+(0..n1+n2-1),p2+(0..n2-1));
3   @ requires is_perm(p1,n1) ∧ is_perm(p2,n2);
4   @ assigns p1[n1..n1+n2-1];
5   @ ensures is_perm(p1,n1+n2);
6   @ ensures is_sum_inplace{Pre,Post}(p1,p2,n1,n1+n2); */
7 void sum_inplace(int p1[], int p2[], int n1, int n2) {
8  int i;
9  /*@ loop invariant n1 ≤ i ≤ n1+n2;
10   @ loop invariant is_sum_inplace{Pre,Here}(p1,p2,n1,i);
11   @ loop invariant is_fct(p1,i,n1+n2) ∧ is_linear(p1,i);
12   @ loop assigns i, p1[n1..n1+n2-1];
13   @ loop variant n1+n2-i; */
14 for (i = n1; i < n1+n2; i++) p1[i] = p2[i-n1]+n1;
15 }
```

Listing 4.14 – Fonction `sum_inplace` annotée.

La spécification et la vérification par preuve des fonctions `sum_inplace` et `remove_max_inplace`, qui modifient aussi le tableau courant, nécessitent également des prédicats, présentés dans le listing 4.12. Leurs spécifications sont présentées respectivement dans les listings 4.13 et 4.14.

4.3 Propriétés de l'insertion

Dans [Baril, 2007], J.-L. Baril considère deux opérations élémentaires sur une permutation p de taille n : l'insertion de la valeur n après une valeur i dans un cycle, et l'ajout du point fixe n. Il affirme alors que toute permutation de taille $n \geq 2$ peut être obtenue de façon unique par l'une de ces deux opérations. Nous démontrons formellement dans cette partie cette affirmation pour l'opération $insert$, qui regroupe ces deux opérations (modulo le fait qu'elle insère la valeur n avant une valeur i dans un cycle, et non après). Plus précisément, nous prouvons que l'opération $insert$, associée à la permutation vide, permet de générer librement et complètement toutes les permutations. Ainsi, nous montrons l'existence d'un type inductif pour les permutations, dont les constructeurs sont la permutation vide et l'opération $insert$ (munie d'un argument entier et d'une contrainte sur cet argument). Cet aspect du sujet ne sera pas approfondi ici. Pour établir cette preuve formelle, nous montrons l'injectivité de l'opération $insert$ (qui correspond à la liberté de la construction) dans la partie 4.3.1, et sa surjectivité (qui correspond à la complétude) dans la partie 4.3.2.

4.3.1 Injectivité de l'insertion

Afin d'établir que l'opération d'insertion *insert*, associée à la permutation vide, est un constructeur de permutations, il convient tout d'abord de montrer l'injectivité de l'insertion. L'injectivité de l'opération *insert* se caractérise par la propriété suivante :

$$\begin{aligned}&\forall n, i_1, i_2, 0 \le i_1 \le n \wedge 0 \le i_2 \le n \Rightarrow \\ &\quad \forall p_1, p_2 \in S_n, insert(p_1, i_1, n) = insert(p_2, i_2, n) \Rightarrow (i_1 = i_2 \wedge p_1 = p_2).\end{aligned} \tag{4.1}$$

Cette propriété stipule que si une permutation de taille $n+1$ a été obtenue de deux manières par insertion de la valeur n dans deux permutations de taille n, alors ces deux permutations sont égales et la valeur n a été insérée à la même position. La contraposée de cette propriété s'énonce plus simplement : "l'insertion de la valeur n dans deux permutations de longueur n différentes ou à des positions différentes donne des permutations de longueur $n+1$ différentes".

Cette propriété peut être montrée à l'aide de la fonction `inj_insert` présentée dans le listing 4.15. Cette fonction construit deux permutations q_1 et q_2, résultats de l'application de la fonction *insert* à deux permutations p_1 et p_2 avec respectivement les entiers i_1 et i_2.

```
1  /*@ requires 0 ≤ i1 ≤ n ∧  0 ≤ i2 ≤ n ∧  \valid(p1+(0..n-1));
2    @ requires \valid(p2+(0..n-1)) ∧  \valid(q1+(0..n)) ∧  \valid(q2+(0..n));
3    @ requires \separated(q1+(0..n),q2+(0..n),p1+(0..n-1),p2+(0..n-1));
4    @ requires is_perm(p1,n) ∧  is_perm(p2,n);
5    @ assigns q1[0..n],q2[0..n];
6    @ ensures is_eq(q1,q2,n+1) ⇒  (i1 == i2 ∧  is_eq(p1,p2,n)); */
7  void inj_insert(int p1[], int p2[], int q1[], int q2[], int i1, int i2, int n) {
8   insert(p1,i1,q1,n);
9   insert(p2,i2,q2,n);
10 }
```

Listing 4.15 – Fonction annotée montrant l'injectivité de la fonction `insert`.

Cette fonction est munie d'un contrat ACSL exprimant la propriété (4.1). Avec les préconditions lignes 1-4, la postcondition ligne 6 est prouvée automatiquement et établit la propriété (4.1), et par conséquent l'injectivité de la fonction *insert*.

```
1 /*@ lemma inj_insert: ∀ ℤ n,i1,i2; (0 ≤ i1 ≤ n ∧  0 ≤ i2 ≤ n) ⇒  ∀ int *p1,*q1,*p2,*q2;
2   @  (\valid(p1+(0..n-1)) ∧  \valid(p2+(0..n-1)) ∧  \valid(q1+(0..n)) ∧  \valid(q2+(0..n))
3   @  ∧  \separated(q1+(0..n),q2+(0..n),p1+(0..n-1),p2+(0..n-1)) ∧  is_perm(p1,n)
4   @  ∧  is_perm(p2,n) ∧  is_insert(q1,p1,i1,n) ∧  is_insert(q2,p2,i2,n) ∧  is_eq(q1,q2,n+1))
5   @  ⇒  (i1 == i2 ∧  is_eq(p1,p2,n)); */
```

Listing 4.16 – Lemme montrant l'injectivité de la fonction `insert`.

Nous pouvons également exprimer directement cette propriété par le lemme ACSL `inj_insert` présenté dans le listing 4.16, à l'aide du prédicat ACSL `is_insert` (équivalent à la fonction `insert`).

Ce lemme est l'exacte traduction en ACSL de la propriété (4.1). Sa preuve automatique par les solveurs SMT est immédiate et établit cette propriété.

4.3.2 Surjectivité de l'insertion

La surjectivité de l'opération *insert* s'exprime par la propriété

$$\forall n, n \geq 0 \Rightarrow \forall q \in S_{n+1}, \exists p \in S_n, \exists i, 0 \leq i \leq n \wedge insert(p,i) = q. \tag{4.2}$$

signifiant que toute permutation q de taille $n+1$ est le résultat de l'application de l'opération $insert$ sur une permutation p de taille n pour un certain entier i entre 0 et n. Le lemme ACSL `surj_insert`, présenté dans le listing 4.17, permet de prouver formellement cette propriété.

```
1 /*@ lemma surj_insert: ∀ ℤ n; n ≥ 0⇒ ∀ int *p,*q,*r;
2   @  (\valid(p+(0..n-1)) ∧ \valid(q+(0..n)) ∧ \valid(r+(0..n))
3   @ ∧ \separated(r+(0..n),q+(0..n),p+(0..n-1)) ∧ is_perm(q,n+1)
4   @ ∧ is_remove(p,q,n,n) ∧ is_insert(r,p,q[n],n))⇒ is_eq(q,r,n+1); */
```

Listing 4.17 – Lemme permettant de prouver la surjectivité de la fonction `insert`.

En effet, ce lemme établit qu'à partir de toute permutation $q \in S_{n+1}$, il existe (i) une permutation $p \in S_n$ obtenue par application de l'opération de retrait du maximum $remove_max$ et (ii) un entier i égal à $q[n]$, tels que l'insertion de n avant i dans p fournit une permutation (notée r dans le lemme) égale à la permutation q.

La preuve formelle de ce lemme, effectuée automatiquement, montre la surjectivité de l'opération $insert$. Notons qu'une fonction munie d'un contrat ACSL aurait également pu être utilisée ici, comme pour l'injectivité.

Associé à la preuve de l'injectivité de l'opération $insert$, nous avons établi formellement que l'opération d'insertion $insert$, associé à la permutation vide, génère librement et complètement les permutations. La démarche adoptée ici est particulière : un résultat mathématique est démontré par le biais de preuves automatiques de lemmes ACSL (ou de fonctions munies d'un contrat ACSL).

4.4 SYNTHÈSE ET PERSPECTIVES

Nous avons implémenté en C quatre opérations sur les permutations, spécifié formellement leur comportement et démontré automatiquement que ces implémentations étaient conformes à leur spécification. Nous avons en outre montré formellement que l'opération d'insertion est un constructeur des permutations.

La table 4.1 donne des statistiques sur ces preuves formelles. La première colonne donne le nom de la fonction ou du lemme prouvé. Le nombre de lignes de code C et le nombre de lignes d'ACSL sont donnés respectivement dans les colonnes 2 et 3. La correction de ces programmes a été prouvée automatiquement avec Frama-C et son greffon WP. La quatrième colonne indique le nombre de conditions de vérification (buts) générés par WP, et la cinquième colonne donne la durée de la preuve de ces buts par les solveurs Alt-Ergo, CVC3 et CVC4, en secondes. La durée allouée à chaque solveur pour chaque obligation de preuve a été étendue de 10 secondes (valeur par défaut) à une minute.

Ces preuves confirment l'intuition qu'une formalisation des permutations en tant que fonctions bijectives sur les entiers est bien adaptée aux raisonnements automatiques.

Une perspective serait d'adapter le présent travail à d'autres opérations sur les permutations, notamment celle de [Baril, 2007], afin de compléter cette formalisation papier en fournissant une formalisation machine vérifiée automatiquement.

Lemme ou fonction	C	ACSL	buts	durée (s)
Fonction insert	4	19	19	25.01
Fonction remove_max	2	17	16	12.51
Fonction sum	2	18	18	19.09
Fonction composition	2	18	17	7.66
Fonction insert_inplace	4	24	20	5.32
Fonction remove_max_inplace	2	21	19	6.65
Fonction sum_inplace	2	21	19	7.19
Fonction inj_insert	3	12	12	.30
Lemme insert2perm	0	3	-	.34
Lemme inj_insert	0	5	-	.34
Lemme surj_insert	0	4	-	.34

TABLE 4.1 – Résultats de vérification des opérations sur les permutations.

La partie de ce travail incluant les fonctions `insert`, `insert_inplace` et `sum` a fait l'objet d'une communication lors de la conférence AFADL en 2016 [Genestier et al., 2016]. Les opérations détaillées dans ce chapitre joueront un rôle important et seront utilisées dans plusieurs chapitres de ce mémoire, notamment les chapitres 8, 9 et 10.

Dans la suite de ce mémoire, nous allons tout d'abord nous intéresser à la génération séquentielle des cartes et hypercartes étiquetées, dans le chapitre 5.

5

Génération séquentielle de cartes combinatoires étiquetées

La bibliothèque `enum` présentée dans le chapitre 3 contient des générateurs qui vont nous servir de briques de base pour générer facilement et rapidement différentes familles d'(hyper)cartes étiquetées, par filtrage. Cette bibliothèque de générateurs de cartes nous permettra alors de visualiser ces objets, et de concevoir et valider des conjectures sur certaines familles de cartes, par test exhaustif borné (défini dans la partie 2.2.2.1 du chapitre 2).

Ce chapitre s'appuie fortement sur les définitions de cartes et hypercartes étiquetées données dans le chapitre 2.

L'un des ingrédients d'une carte combinatoire ordinaire est une involution sans point fixe. Le générateur d'involutions sans point fixe FFI présenté dans le chapitre 3 est peu efficace, car il procède par filtrage parmi les involutions. Dans le but de générer plus efficacement les cartes ordinaires, nous avons conçu un générateur original d'involutions sans point fixe. Il est présenté dans la partie 5.1, ainsi que sa vérification déductive. D'autre part, les définitions d'(hyper)cartes du chapitre 2 utilisent la notion de paire de permutations agissant transitivement sur leur support. Nous avons donc conçu un algorithme caractérisant cette action transitive, décrit dans la partie 5.2. Cet algorithme permet d'implémenter une génération exhaustive bornée de différents types d'(hyper)cartes étiquetées. Nous avons d'abord mis au point un générateur générique de couples de tableaux, présenté dans la partie 5.3. Ces couples sont ensuite filtrés par la propriété de transitivité grâce à un second générateur générique présenté dans la partie 5.4. La partie 5.5 décrit l'application de ces générateurs à différentes familles de permutations, afin d'obtenir des générateurs séquentiels d'(hyper)cartes étiquetées. Pour simplifier la définition d'une carte ordinaire et ainsi faciliter sa formalisation et sa génération, nous introduisons dans la partie 5.6 la notion de carte locale, qui jouera un rôle important dans la suite de ce mémoire. Son mode de génération est aussi détaillé. Enfin, des résultats expérimentaux et une synthèse de ce chapitre sont respectivement donnés dans les parties 5.7 et 5.8.

Les fichiers sources des générateurs présentés dans ce chapitre sont disponibles dans les répertoires `generation/effi/` et `generation/generic/` de la bibliothèque `enum`.

5.1 Générateur efficace d'involutions sans point fixe

Générer des involutions sans point fixe (ou FFI pour Fixpoint Free Involution) par filtrage parmi les involutions n'est pas une méthode optimale si l'on souhaite générer des cartes ordinaires de la taille la plus grande possible. Il nous a donc semblé important de disposer d'un générateur plus efficace de FFIs. Nous avons conçu un tel générateur, écrit sa spécification formelle et prouvé sa correction, comme nous l'avons fait pour différents générateurs de tableaux dans le chapitre 3. Ce générateur est décrit dans la partie 5.1.1. Sa spécification formelle et sa vérification par preuve sont détaillées dans la partie 5.1.2. Des tests complémentaires et des résultats expérimentaux sont donnés dans la partie 5.1.3.

5.1.1 Algorithme

Des algorithmes listant toutes les involutions sans point fixe existent dans la littérature [Walsh, 2001, Prissette, 2010, Vajnovszki, 2010]. Cependant, ces algorithmes étant récursifs, les implémenter ne serait pas cohérent avec notre démarche. En effet, les outils de vérification existants que nous utilisons ne permettent pas (encore) d'effectuer des preuves automatiques de programmes récursifs. Nous avons donc mis au point un générateur séquentiel itératif de FFIs, nommé EFFI.

Cet algorithme utilise une involution sans point fixe particulière.

Définition 18 : Involution sans point fixe locale

Soit $n > 0$. On appelle involution sans point fixe locale de taille $2n$ la permutation sur $\{0, \dots, 2n-1\}$ qui échange les entiers $2i$ et $2i+1$ pour tout i tel que $0 \le i < n$.

Cette involution sans point fixe est dite locale car elle échange une valeur de son domaine avec sa voisine immédiate dans l'ordre sur les entiers. Dans la suite, le plus souvent, la taille de l'involution sans point fixe locale considérée n'est pas précisée, car elle se déduit facilement du contexte. Elle est notée

$$\begin{pmatrix} 0 & 1 & \dots & 2n-2 & 2n-1 \\ 1 & 0 & \dots & 2n-1 & 2n-2 \end{pmatrix} = 1\ 0\ \dots\ (2n-1)\ (2n-2) = (0\ 1)\ \dots\ ((2n-2)\ (2n-1)).$$

Cette involution sans point fixe locale sera également utilisée pour définir un type de carte original dans la partie 5.6 de ce chapitre.

Le générateur séquentiel EFFI est composé de deux fonctions `first_effi` et `next_effi`. Contrairement aux générateurs présentés dans le chapitre 3, ce générateur ne construit pas les FFIs dans l'ordre lexicographique. La première FFI de taille **paire** n, générée par la fonction `first_effi` non reproduite ici, est la FFI

$$\begin{pmatrix} 0 & 1 & \dots & n-2 & n-1 \\ n-1 & n-2 & \dots & 1 & 0 \end{pmatrix} = (n-1)\ (n-2)\ \dots\ 1\ 0 = (0\ (n-1))\ (1\ (n-2))\ \dots$$

qui échange les éléments i et $n-1-i$ pour $0 \le i < n$.

La fonction de succession `next_effi`, donnée dans le listing 5.1, génère les FFIs de taille n suivantes, jusqu'à la dernière, qui est la FFI locale

$$1\ 0\ \dots\ (n-2)\ (n-1) = (0\ 1)\ \dots\ ((n-1)\ (n-2)).$$

```c
1  int next_effi(int a[], int n) {
2   int tmp1,tmp2,t,i,rev = 0;
3
4   while (rev < n ∧ a[rev] == rev+1) rev += 2;
5   if (rev ≥ n) return 0;
6
7   t = a[rev];
8   tmp1 = a[t-1]; a[t-1] = rev; a[t] = tmp1;
9   tmp2 = t; a[rev] = a[tmp1]; a[tmp1] = tmp2;
10
11  for (i = rev; i > 0; i -= 2) {
12   for (j = i-1; j < n-1; j++) a[j] = a[j+1] - 1;
13   a[i-2] = n-1; a[n-1] = i-2;
14  }
15  return 1;
16 }
```

Listing 5.1 – Fonction `next_effi`.

La fonction `next_effi` commence par rechercher l'index de révision rev du tableau a, qui est le plus petit index pair tel que $a[rev]$ est différent de $rev + 1$, c'est-à-dire le plus petit index à partir duquel le préfixe de la FFI courante diffère de la FFI locale (ligne 4). Si l'index de révision est supérieur ou égal à n, la FFI locale de taille n est atteinte et la fonction retourne 0 (ligne 5). Sinon, les valeurs $a[a[rev]]$ et $a[a[rev] - 1]$ sont échangées (ligne 8), puis les valeurs $a[rev]$ et $a[a[a[rev] - 1]]$ sont échangées (ligne 9). Si l'index de révision est égal à 0, la prochaine FFI est construite et la fonction retourne 1. Dans le cas contraire, la fonction modifie toutes les valeurs de a à l'aide de $rev/2$ itérations de la boucle lignes 11-14, principalement par des décalages de valeurs.

Exemple [14]. *La table 5.1 présente deux appels successifs de la fonction* `next_effi` *sur la FFI* $2\ 3\ 0\ 1\ 7\ 8\ 9\ 4\ 5\ 6 = (0\ 2)\ (1\ 3)\ (4\ 7)\ (5\ 8)\ (6\ 9)$ *de taille 10.*

Action de la ligne	Notation sur une ou deux ligne(s) de a	Notation cyclique de a
-	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 3 & 0 & 1 & 7 & 8 & 9 & 4 & 5 & 6 \end{pmatrix}$	(0 2) (1 3) (4 7) (5 8) (6 9)
8	2 **0** **3** 1 7 8 9 4 5 6	(0 2 3 1) (4 7) (5 8) (6 9)
9	**1** 0 3 **2** 7 8 9 4 5 6	(0 1) (2 3) (4 7) (5 8) (6 9)
-	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \mathit{1} & \mathit{0} & \mathit{3} & \mathit{2} & 7 & 8 & 9 & 4 & 5 & 6 \end{pmatrix}$	(0 1) (2 3) (4 7) (5 8) (6 9)
8	*1 0 3 2* 7 8 **4** **9** 5 6	(0 1) (2 3) (4 7 9 6) (5 8)
9	*1 0 3 2* **6** 8 4 9 5 **7**	(0 1) (2 3) (4 6) (5 8) (7 9)
12-13	*1 0* **9 5 7 3 8 4 6 2**	(0 1) (2 9) (3 5) (4 7) (6 8)
12-13	**9 8 4 6 2 7 3 5 1 0**	(0 9) (2 4) (3 6) (5 7) (1 8)

TABLE 5.1 – Trace d'exécution de la fonction `next_effi` sur les FFI 2 3 0 1 7 8 9 4 5 6 et 1 0 3 2 7 8 9 4 5 6.

La première colonne donne le numéro de la ligne du listing 5.1 comportant les instructions permettant d'obtenir, à partir de la permutation précédente, la permutation dont la notation sur une ou deux ligne(s) et la notation cyclique sont données respectivement dans les colonnes 2 et 3. Les valeurs modifiées sont alors indiquées en gras, tandis que le préfixe constitué de la FFI locale est indiqué en italique. Pour alléger la table, la notation sur deux lignes comportant les indices n'est donnée que pour montrer la FFI initiale.

Lors du premier appel, la condition de la boucle ligne 4 n'étant jamais vérifiée, l'indice de révision rev *de cette FFI est égal à* 0*. La FFI suivante* $1\ 0\ 3\ 2\ 7\ 8\ 9\ 4\ 5\ 6$ *s'obtient alors*

directement à l'aide des deux échanges de valeurs $3 \leftrightarrow 0$ *(aux indices 1 et 2) et* $2 \leftrightarrow 1$ *(aux indices 0 et 3).*

Le second appel de la fonction `next_effi`*, à partir de la FFI* $1\ 0\ 3\ 2\ 7\ 8\ 9\ 4\ 5\ 6$*, donne un indice de révision* rev *égal à* 4*, qui est la longueur de la FFI locale* $1\ 0\ 3\ 2 = (0\ 1)\ (2\ 3)$ *constituant un préfixe de cette FFI. La FFI résultant des deux échanges de valeurs* $9 \leftrightarrow 4$ *et* $7 \leftrightarrow 6$ *est* $1\ 0\ 3\ 2\ 6\ 8\ 4\ 9\ 5\ 7$*. Il s'ensuit deux itérations de la boucle ligne 11. Lors de la première exécution de la boucle ligne 12, avec* $i = 4$*, Les valeurs* $6\ 8\ 4\ 9\ 5\ 7$ *de* $a[4..9]$ *sont décrémentées puis décalées vers la gauche pour donner les nouvelles valeurs* $5\ 7\ 3\ 8\ 4\ 6$ *de* $a[3..8]$*. Les insertions des valeurs* 9 *et* 2 *aux indices* 2 *et* 9 *(ligne 13) permettent d'obtenir la FFI* $1\ 0\ 9\ 5\ 7\ 3\ 8\ 4\ 6\ 2$*. La deuxième exécution de la boucle ligne 12, avec* $i = 2$*, effectue un nouveau décalage : les valeurs* $9\ 5\ 7\ 3\ 8\ 4\ 6\ 2$ *de* $a[2..9]$ *permettent d'obtenir les nouvelles valeurs* $8\ 4\ 6\ 2\ 7\ 3\ 5\ 1$ *de* $a[1..8]$*. Finalement la nouvelle FFI* $9\ 8\ 4\ 6\ 2\ 7\ 3\ 5\ 1\ 0$ *s'obtient après insertion des valeurs* 9 *et* 0 *aux indices* 0 *et* 9 *(ligne 13).*

Globalement, dans le listing 5.1, il est bien évident que les phases constituées d'échanges de valeurs (lignes 7-9 et 13) préservent la propriété d'être une FFI. La phase de décalage de la ligne 12 décrémente à la fois des valeurs et des indices, donc ne modifie pas la longueur de chaque cycle et préserve ainsi la propriété d'être une FFI, même si on l'itère (ligne 11). Globalement, la propriété d'être une FFI est donc préservée par l'algorithme entier. Pour conforter ce constat, nous avons vérifié formellement la correction de la fonction `next_effi`, comme détaillé dans la partie suivante.

5.1.2 Spécification formelle et preuve

Pour caractériser une FFI, nous définissons les prédicats ACSL présentés dans le listing 5.2.

```
1  /*@ predicate is_fct(int *a, ℤ b, ℤ c) = ∀ ℤ i; 0 ≤ i < b⇒ 0 ≤ a[i] < c;
2    @ predicate is_linear(int *a, ℤ b, ℤ c) =
3    @  ∀ ℤ j; b ≤ j < c⇒ (∀ ℤ k; j < k < c⇒ (a[j] == a[k]⇒ j == k));
4    @ predicate is_perm(int *a, ℤ b) = is_linear(a,0,b) ∧ is_fct(a,b,b);
5    @ predicate is_inv(int *a, ℤ b, ℤ c) = ∀ ℤ i; b ≤ i < c⇒ a[a[i]] == i;
6    @ predicate is_invol(int *a, ℤ b) = is_perm(a,b) ∧ is_inv(a,0,b);
7    @ predicate is_ff(int *a, ℤ b, ℤ c) = ∀ ℤ i; b ≤ i < c⇒ a[i] ≠ i;
8    @ predicate is_ffi(int *a, ℤ b) = is_ff(a,0,b) ∧ is_invol(a,b);
9    @ predicate is_local(int *a, ℤ b) =
10   @  ∀ int j; 0 ≤ j < b⇒ a[j] == (j%2 == 0 ? j+1 : j-1); */
```

Listing 5.2 – Prédicats ACSL pour caractériser une FFI.

Le prédicat `is_ffi` qui caractérise une FFI, présenté ligne 8, est la conjonction des prédicats `is_invol` (ligne 6) caractérisant une involution et `is_ff` caractérisant l'absence de point fixe dans une permutation (ligne 7). Le prédicat `is_invol` est lui-même la conjonction des prédicats `is_perm` (ligne 4) caractérisant une permutation et `is_inv` (ligne 5) caractérisant une involution parmi les permutations. Comme dans le chapitre 4, une permutation de taille n est définie par une endofonction injective sur $\{0, \ldots, n-1\}$. Les prédicats `is_fct`, `is_linear` et `is_perm` sont donc similaires à ceux présentés dans le listing 4.5 du chapitre 4. La propriété caractéristique d'une fonction est exprimée par le prédicat `is_fct` présenté ligne 1. La propriété d'injectivité est exprimée par le prédicat `is_linear` présenté lignes 2-3, qui comporte ici trois paramètres car il n'est pas toujours utilisé à partir de l'indice 0. Enfin, le prédicat `is_local` (lignes 9-10) spécifie la

propriété d'être locale pour une FFI. Ce prédicat sera utilisé pour propager cette propriété aux différentes sous-fonctions composant la fonction `next_effi`.

Pour favoriser sa vérification par preuve, la fonction `next_effi` est décomposée en 4 sous-fonctions, dont on spécifiera précisément le comportement. Cette décomposition et le contrat ACSL de la fonction sont présentés dans le listing 5.3.

```
1  /*@ requires n > 0 ∧ n%2 == 0 ∧ \valid(a+(0..n-1)) ∧ is_ffi(a,n);
2    @ assigns a[0..n-1];
3    @ ensures is_ffi(a,n); */
4  int next_effi(int a[], int n) {
5   int rev = rev_point(a,n);
6   if (rev ≥ n) return 0;
7   swap(a,n,rev);
8   decal_ext(a,n,rev);
9   return 1;
10 }
```

Listing 5.3 – Fonction `next_effi` décomposée en sous-fonctions.

La fonction `rev_point`, présentée dans le listing 5.4 avec son contrat ACSL, effectue la recherche du point de révision rev et le renvoie. La spécification doit notamment assurer que, si cet indice permet d'obtenir une nouvelle FFI (c.a.d. s'il est inférieur à n), les deux échanges de valeurs qui s'ensuivront seront possibles. Les conditions $a[rev] > 0$ et $a[a[rev]-1] \neq a[rev]$ de leur exécution sont vérifiées et assurées par la postcondition ligne 4.

La postcondition ligne 5 assure que le préfixe `a[0..rev-1]` est une FFI locale (éventuellement vide) de taille rev. Cette propriété permet d'effectuer la preuve de la correction de la fonction `decal_ext`. Elle est donc "propagée" dans les contrats des sous-fonctions `swap` et `decal_in` où elle apparaît en tant que précondition et postcondition.

```
1  /*@ requires n > 0 ∧ n%2 == 0 ∧ \valid(a+(0..n-1)) ∧ is_ffi(a,n);
2    @ assigns \nothing;
3    @ ensures \result ≥ 0 ∧ \result%2 == 0;
4    @ ensures \result < n⇒ (a[\result] > 0 ∧ a[a[\result]-1] ≠ a[\result]);
5    @ ensures is_local(a,\result); */
6  int rev_point(int a[], int n) {
7   int rev = 0;
8   /*@ loop invariant 0 ≤ rev ≤ n ∧ rev%2 == 0 ∧ is_local(a,rev);
9     @ loop assigns rev;
10    @ loop variant n-rev; */
11  while (rev < n ∧ a[rev] == rev+1) rev += 2;
12  return rev;
13 }
```

Listing 5.4 – Fonction `rev_point`.

La fonction `swap`, présentée dans le listing 5.5 avec son contrat ACSL, effectue les deux échanges de valeurs, sous les conditions prouvées grâce au contrat de la fonction `rev_point`.

```
1  /*@ requires n > i ≥ 0 ∧ n%2 == 0 ∧ i%2 == 0 ∧ \valid(a+(0..n-1));
2    @ requires a[i] > 0 ∧ a[a[i]-1] ≠ a[i] ∧ is_ffi(a,n) ∧ is_local(a,i);
3    @ assigns a[i],a[a[i]-1],a[a[i]],a[a[a[i]-1]];
4    @ ensures is_ffi(a,n) ∧ is_local(a,i); */
5  void swap(int a[], int n, int i) {
6   int t = a[i];
7   int tmp1 = a[t-1]; a[t-1] = a[t]; a[t] = tmp1;
8   /*@ assert a[a[i]-1] == i ∧ a[a[a[i]-1]] == a[i]; */
9   int tmp2 = t; a[i] = a[tmp1]; a[tmp1] = tmp2;
10  /*@ assert a[a[i]] == i ∧ a[a[a[i]]] == a[i]; */
11 }
```

Listing 5.5 – Fonction `swap`.

Il faut ici s'assurer qu'après ces échanges de valeurs le tableau a représente toujours une FFI et que son préfixe $a[0..rev-1]$ est une FFI locale (postcondition ligne 4). Cette tâche étant complexe, il convient de spécifier précisément quels sont les éléments de a qui sont modifiés (ligne 3), et d'aider les prouveurs à prouver la conservation de la propriété d'involution à l'aide des assertions lignes 8 et 10.

La fonction `decal_ext`, présentée dans le listing 5.6 lignes 26-37, effectue les $rev/2$ décalages de valeurs dans le cas où $rev > 0$, par des appels successifs à la fonction `decal_in`, présentée lignes 1-24 de ce listing. La fonction `decal_in` est la plus délicate à spécifier. Nous devons nous assurer qu'en sortie le tableau a représente toujours une FFI dont le préfixe $a[0..rev-3]$ est local suite à la modification des valeurs de a entre les indices $rev-2$ et $n-1$ inclus. Dans ce but, nous utilisons des prédicats additionnels pour spécifier finement le comportement de la boucle ligne 18. Ces prédicats sont présentés dans le listing 5.7.

```
1  /*@ requires n > rev > 1 ∧ n%2 == 0 ∧ rev%2 == 0 ∧ \valid(a+(0..n-1));
2    @ requires is_ffi(a,n) ∧ is_local(a,rev);
3    @ assigns a[rev-2..n-1];
4    @ ensures is_ffi(a,n);
5    @ ensures is_local(a,rev-2); */
6  void decal_in(int a[], int n, int rev) {
7
8   /*@ loop invariant rev-1 ≤ j ≤ n-1;
9     @ loop invariant l1: is_decal_in{Pre,Here}(a,rev-1,j);
10    @ loop invariant l2: is_eq_gt{Pre,Here}(a,j,n);
11    @ loop invariant l3: is_fct(a,j,n-1);
12    @ loop invariant l4: is_linear(a,rev-1,j);
13    @ loop invariant l5: is_gt(a,rev-1,j,rev-2);
14    @ loop invariant l6: is_ff(a,rev-1,j);
15    @ loop invariant l7: is_inv_decal(a,rev-1,j);
16    @ loop assigns j,a[rev-1..n-2];
17    @ loop variant n-1-j; */
18  for (int j = rev-1; j < n-1; j++) a[j] = a[j+1] - 1;
19  //@ assert is_inv(a,rev-1,n-1);
20  a[rev-2] = n-1; a[n-1] = rev-2;
21  //@ assert is_fct(a,n,n);
22  //@ assert is_linear(a,0,n);
23  //@ assert is_inv(a,0,n);
24 }
25
26 /*@ requires n > rev ≥ 0 ∧ n%2 == 0 ∧ rev%2 == 0 ∧ \valid(a+(0..n-1));
27   @ requires is_ffi(a,n) ∧ is_local(a,rev);
28   @ assigns a[0..n-1];
29   @ ensures is_ffi(a,n); */
30 void decal_ext(int a[], int n, int rev) {
31
32  /*@ loop invariant 0 ≤ k ≤ rev ∧ k%2 == 0;
33    @ loop invariant is_ffi(a,n) ∧ is_local(a,k);
34    @ loop assigns k,a[0..n-1];
35    @ loop variant k; */
36  for (int k = rev; k > 1; k-=2) decal_in(a,n,k);
37 }
```

Listing 5.6 – Fonctions `decal_ext` et `decal_in`.

```
1 /*@ predicate is_gt(int *a, ℤ b, ℤ c, ℤ d) = ∀ ℤ j; b ≤ j < c⇒ a[j] > d;
2   @ predicate is_inv_decal(int *a, ℤ b, ℤ c) =
3   @   ∀ ℤ j; (b ≤ j < c ∧ a[j] < c)⇒ a[a[j]] == j;
4   @ predicate is_decal_in{L1,L2}(int *a, ℤ b, ℤ c) =
5   @   ∀ ℤ j; b ≤ j < \at(c,L2)⇒ \at(a[j],L2) == \at(a[j+1],L1)-1;
6   @ predicate is_eq_gt{L1,L2}(int *p, ℤ b, ℤ c) =
7   @   ∀ ℤ j; b ≤ j < c⇒ \at(p[j],L2) == \at(p[j],L1); */
```

Listing 5.7 – Prédicats ACSL complémentaires pour la fonction `decal_in`.

Le prédicat `is_gt` spécifie que toutes les valeurs d'un tableau a entre deux bornes données sont supérieures à une valeur donnée. Le prédicat `is_inv_decal` assure que la propriété d'involution est vérifiée dans un tableau a entre deux bornes données, aux indices des valeurs strictement inférieures à la deuxième borne. D'autre part, dans cette situation, il est nécessaire de comparer différentes valeurs du même tableau dans deux états différents de l'exécution du programme, comme nous l'avons déjà vu dans les chapitres 3 et 4. Nous utilisons encore ici des prédicats hybrides, avec des labels. Le prédicat `is_decal_in` paraphrase le code de la boucle ligne 18 en précisant les labels d'évaluation des valeurs du tableau a. Il faut également s'assurer que lors de l'exécution de la boucle, la partie du tableau située après l'indice courant de boucle n'est pas modifiée. Le prédicat `is_eq_gt`, introduit dans la partie 4.2.2, spécifie ainsi qu'à chaque itération de la boucle, les valeurs situées après l'indice courant de boucle sont égales dans les états `L1` et `L2`.

Ces prédicats complémentaires, ainsi que ceux présentés dans le listing 5.2, nous permettent de spécifier à l'aide d'invariants de boucle que lors de l'exécution de la boucle de la fonction `decal_in`, jusqu'à l'indice courant, le tableau a :

- a toutes ses valeurs inférieures à $n - 1$ (ligne 11),
- a toutes ses valeurs différentes depuis l'indice $rev - 1$ (ligne 12),
- a toutes ses valeurs supérieures à $rev - 2$ depuis l'indice $rev - 1$ (ligne 13),
- n'a pas de point fixe depuis l'indice $rev - 1$ (ligne 14), et
- a des valeurs vérifiant la propriété d'involution depuis l'indice $rev - 1$ (ligne 15). Cet invariant de boucle permet d'établir l'assertion ligne 19 établissant que les éléments de la partie $a[rev - 1..n - 2]$ du tableau a vérifient la propriété d'involution.

En complément, nous spécifions ligne 10 que les valeurs situées après l'indice courant de boucle sont égales dans les états `Pre` et `Here`, et ligne 9 que chaque valeur située avant l'indice courant de boucle est égale à la valeur de l'indice successeur à l'état `Pre` décrémentée de 1.

Suite à cette boucle modifiant les valeurs $a[rev - 1..n - 2]$, les valeurs d'indices $rev - 2$ et $n - 1$ sont modifiées ligne 20. A l'issue de ces deux affectations, nous ajoutons des assertions permettant aux prouveurs de prouver la postcondition ligne 4. A cet effet, nous établissons que le tableau a représente une fonction sur $\{0, \ldots, n - 1\}$ grâce à l'assertion

```
assert is_fct(a,n,n);
```

prouvée grâce à l'invariant ligne 11. Nous établissons également que cette fonction est injective grâce à l'assertion

```
assert is_linear(a,0,n);
```

prouvée grâce aux invariants lignes 13 et 14. Enfin nous établissons que les éléments du tableau a vérifient la propriété d'involution grâce à l'assertion

```
assert is_inv(a,0,n);
```

prouvée grâce à la précédente assertion ligne 19. L'invariant ligne 15 complète ces spécifications qui permettent de prouver que le tableau en sortie représente bien une FFI. La postcondition

```
ensures is_local(a,rev-2);
```

est prouvée grâce à la précondition

```
requires is_local(a,rev);
```

et à la clause

```
assigns a[rev-2..n-1];
```

Finalement, les postconditions de la fonction `decal_in` permettent d'établir la postcondition

```
ensures is_ffi(a,n);
```

de la fonction `decal_ext` grâce à l'invariant ligne 33. Cette dernière postcondition permet ainsi de prouver la correction de la fonction `next_effi`.

Lors de la preuve formelle de la fonction `next_effi`, les solveurs Alt-Ergo, CVC3 et CVC4 déchargent 123 obligations de preuves en 5 minutes 35 secondes. La durée de calcul allouée aux prouveurs pour décharger les obligations de preuve a ici été portée à 10 minutes. La durée élevée de preuve reflète la complexité de la spécification formelle, plus coûteuse en terme d'effort de preuve que celle des générateurs présentés dans le chapitre 3. Elle indique également que l'on approche ici des limites des capacités des prouveurs actuels.

5.1.3 VALIDATION ET RÉSULTATS EXPÉRIMENTAUX

Ce générateur ne produit pas les FFIs dans l'ordre lexicographique. Aussi la propriété de progression, détaillée dans le chapitre 3, n'a pas pu être vérifiée. Nous avons en revanche validé la propriété d'exhaustivité de ce générateur par génération exhaustive bornée, en comptant le nombre de FFIs générées de taille $n = 2k$ et en le comparant au k-ième terme de la séquence A001147 de l'OEIS [The OEIS Foundation Inc., 2010]. Cette validation a été effectuée jusqu'à la taille $n = 24$ en 1 heure 8 minutes, produisant $316\ 234\ 143\ 225$ FFIs.

Sans faire d'étude de complexité algorithmique, nous avons comparé les temps de génération de FFIs du générateur EFFI avec ceux du générateur par filtrage FFI (présenté dans le chapitre 3), mais aussi avec ceux d'implémentations de deux autres algorithmes de la littérature [Prissette, 2010, Vajnovszki, 2010] générant les FFIs.

La figure 5.1 représente graphiquement les durées de génération pour chaque générateur de FFIs, en fonction de la demi-cardinalité de leur support, sur une échelle logarithmique en ordonnée. Dans cette figure, le générateur issu de [Prissette, 2010] est noté PRI et celui issu de [Vajnovszki, 2010] est noté VAJ. Nous mettons en évidence ici le gain important d'efficacité du générateur efficace EFFI par rapport au générateur par filtrage FFI. Nous pouvons également visualiser que l'efficacité du générateur EFFI et très proche de celle du générateur VAJ.

5.2 TRANSITIVITÉ D'UNE PAIRE DE PERMUTATIONS

Dans le chapitre 2, les définitions d'(hyper)cartes sont fondées sur l'action transitive d'un groupe sur un ensemble. Cette action est transitive si deux éléments quelconques de l'ensemble peuvent être envoyés l'un sur l'autre par l'action d'un élément du groupe. Dans les explications de cette partie, nous nous restreignons au cadre d'un groupe généré par deux permutations, qui est notre cadre d'étude. Nous notons ici p_1 et p_2 les deux permutations considérées.

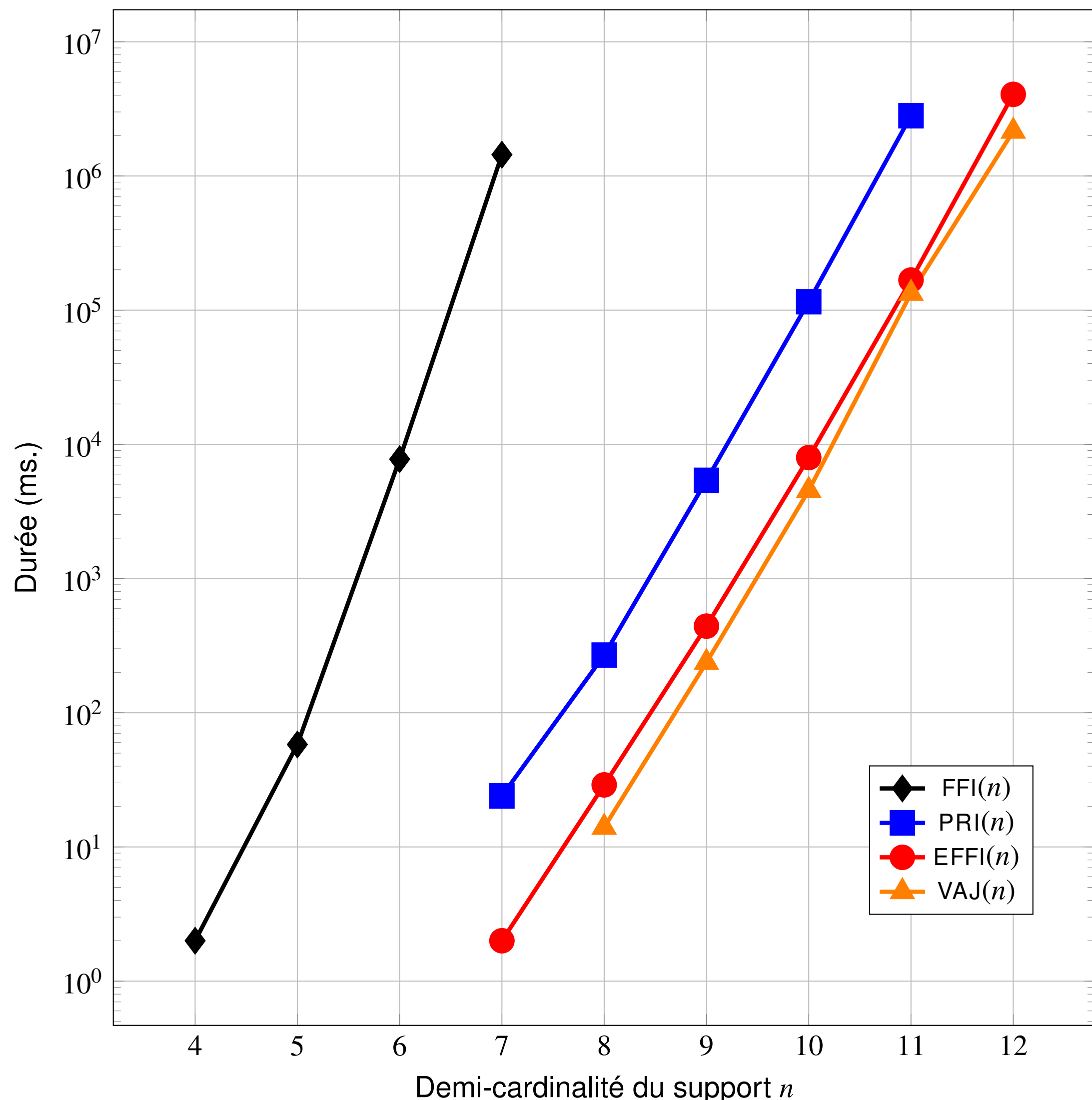


FIGURE 5.1 – Durées de génération de FFIs en fonction de la demi-cardinalité de leur support.

Le groupe engendré par deux permutations est constitué de tous les produits possibles de ces deux permutations et de leurs inverses. Or, si $x \in D = \{0, \ldots, n-1\}$ appartient à un cycle de taille $k \leq n$ d'une permutation $p \in S_n$, le prédécesseur de x dans ce cycle peut s'atteindre par $k-1$ applications de p à partir de x, c.a.d. $p^{-1}(x) = p^{k-1}(x)$. Nous pouvons donc ici ignorer l'action des inverses des permutations et réduire l'action transitive à une suite finie d'applications des deux permutations sur les éléments de l'ensemble D.

Dans ce contexte, l'action transitive se traduit par le fait que tout brin de D peut être relié à tout autre brin de D par une suite finie d'applications de p_1 et p_2. Nous pouvons simplifier cette caractérisation en considérant un brin quelconque comme "brin initial" à partir duquel il suffit d'atteindre tous les autres brins. La caractérisation de cette action

transitive nécessite alors un parcours des brins par des applications successives des deux permutations à partir de ce brin arbitraire.

Ce problème se rapproche d'une thématique déjà largement explorée : le parcours d'un graphe étiqueté. Il existe différents algorithmes classiques pour explorer un graphe en informatique et en théorie des graphes. L'un des plus utilisés est nommé parcours en profondeur d'abord (DFS, pour *Depth-First Search*) [Knuth, 1997]. Cet algorithme effectue un parcours des sommets d'un graphe, en marquant chaque sommet visité. Dans sa version itérative, cet algorithme utilise une pile pour stocker les sommets à visiter. Le sommet initial est placé dans la pile, puis, tant que la pile n'est pas vide :

— Le dernier sommet empilé est dépilé.
— S'il n'est pas visité, il est marqué visité, et tous ses sommets adjacents sont empilés.

Dans [Jacquot, 2014], A. Jacquot utilise un tel algorithme, sans le décrire, pour parcourir une carte en profondeur d'abord, pour décomposer les cartes comme des objets constructibles, afin d'appliquer des méthodes efficaces pour la génération aléatoire. Cette approche est basée sur le parcours des sommets d'une carte, et non de ses brins.

Dans [Vidal, 2010], S. Vidal fournit notamment un algorithme récursif de renommage des brins de cartes dont chaque sommet possède trois brins incidents. Cet algorithme se rapproche de notre objectif car il agit sur les brins d'une carte, mais nous avons préféré adapter le principe de l'algorithme DFS, sous sa forme itérative, plus proche de notre besoin. En effet, vérifier que tout sommet d'un graphe puisse être relié à tout autre sommet par les arcs d'un graphe, ou que tout brin d'une carte puisse être relié à tout autre brin par l'action de deux permutations, sont deux actions similaires. Ainsi notre algorithme peut reprendre le principe de l'algorithme DFS, mais en parcourant non pas les sommets d'un graphe, mais les brins du support de deux permutations.

Soient $p_1 \in S_n$ et $p_2 \in S_n$ deux permutations de même support. Comme dans l'algorithme DFS, tout brin est soit “inconnu”, soit “visité”. Initialement, seul le brin initial est visité, tous les autres brins sont inconnus. L'algorithme cherche à visiter le maximum de brins, par l'application de p_1 et p_2, et à les compter. Puisque chaque brin peut avoir deux images différentes, selon qu'on lui applique p_1 ou p_2, une recherche itérative linéaire du brin suivant est impossible. On stocke donc temporairement ces images. Un brin ainsi stocké est dit “visité”. On lui applique p_1 et p_2 pour visiter ses voisins, qu'on ajoute au stock, tandis qu'on le retire du stock car il n'est plus utile pour découvrir de nouveaux brins.

```
1  int b_trans(int p1[], int p2[], int n) {
2   int B[n],  // Tableau de statut des brins :
3              // B[i] = 0 si i inconnu, B[i] = 1 si i visite'
4       S[n],  // Pile stockant les brins visite's
5       h = 0, // Hauteur de pile
6       c = 1, // Nombre de brins visite's
7       d;     // Brin courant
8   // initialisations
9   S[h] = n-1; B[n-1] = 1;
10  for (int i = 0; i < n-1; i++) B[i] = 0;
11  // Tant qu'il reste des brins a` visiter et des brins visite's non de'pile's
12  while (c ≠ n ∧ h ≠ -1) {
13   d = S[h]; h--;  // le dernier brin visite' est de'pile'.
14   // Si p1[d] est inconnu, il devient visite'
15   if (B[p1[d]] == 0) { B[p1[d]] = 1; c++; h++; S[h] = p1[d]; }
16   // Si p2[d] est inconnu, il devient visite'
17   if (B[p2[d]] == 0) { B[p2[d]] = 1; c++; h++; S[h] = p2[d]; }
18  }
19  // On indique en sortie si tous les brins sont visite's
20  return (c == n);
```

```
21 }
```

Listing 5.8 – Fonction `b_trans` caractérisant l'action transitive de deux permutations.

L'algorithme est présenté à travers son implémentation sous forme d'une fonction booléenne C `b_trans` reproduite dans le listing 5.8. Cette implémentation prend en paramètre deux permutations p_1 et p_2 ainsi que leur taille n. La fonction `b_trans` utilise les variables locales suivantes :

- — Un tableau d'entiers B contenant le statut de chaque brin i, tel que $B[i]$ a pour valeur 0 si i est inconnu et 1 si i est visité.
- — Une pile S de hauteur h (initialement vide) contenant temporairement les brins visités.
- — Un entier c comptant le nombre de brins visités.
- — Un entier d représentant le brin courant.

Avant d'entrer dans la boucle, le brin initial $n - 1$ (arbitraire) est placé dans la pile S, marqué visité dans B (lignes 9), et tous les autres brins sont initialisés à "inconnu" (ligne 9). L'algorithme explore les images du brin courant par p_1 et p_2 à l'aide d'une boucle (lignes 12-18) tant qu'il reste des brins à visiter et que la pile contient des brins visités. Au sein de la boucle, le dernier brin visité est retiré de la pile S et constitue le nouveau brin courant (ligne 13). Les brins images de ce brin par p_1 et p_2 prennent alors le statut "visité" s'ils étaient inconnus, le nombre de brins visités est alors incrémenté, et ils sont empilés (lignes 15 et 17). Si tous les brins ont été visités, la fonction `b_trans` retourne 1, sinon elle retourne 0 (ligne 20). Cet algorithme suit stricto-sensu le déroulement de l'algorithme DFS décrit plus haut, avec deux optimisations : les itérations de la boucle cessent si tous les brins ont été visités (même s'il en reste dans la pile), et un brin n'est placé dans la pile que s'il est inconnu.

Exemple [15]. *Considérons les permutations* $p_1 = (0\ 4\ 7\ 2)\ (1\ 5\ 3)\ (6)$ *et* $p_2 = (0\ 7)\ (1\ 5)\ (2\ 4\ 3\ 6)$ *de taille 8. Avec* p_1 *et* p_2 *en entrée, la fonction initialise* c, S *et* B,

d	S	B								c
		0	1	2	3	4	5	6	7	
-	7	0	0	0	0	0	0	0	**1**	1
7	0 2	**1**	0	**1**	0	0	0	0	1	3
0	4 2	1	0	1	0	**1**	0	0	1	4
4	3 2	1	0	1	**1**	1	0	0	1	5
3	6 1 2	1	**1**	1	1	1	0	**1**	1	7
6	1 2	1	1	1	1	1	0	1	1	7
1	5 2	1	1	1	1	1	**1**	1	1	8

TABLE 5.2 – Trace d'exécution de la fonction `b_trans` sur les permutations $p_1 = (0\ 4\ 7\ 2)\ (1\ 5\ 3)\ (6)$ et $p_2 = (0\ 7)\ (1\ 5)\ (2\ 4\ 3\ 6)$.

puis déclenche 6 *itérations de la boucle, décrites par la table 5.2. La première ligne de cette table indique que le brin* 7 *est empilé, marqué visité, et qu'il n'existe pas encore de brin courant. Chaque ligne suivante du tableau donne l'état des variables à l'issue de chaque itération de la boucle. La deuxième ligne indique que* 7 *est dépilé et devient le brin courant. Par la suite, le brin* 2 *(image de* 7 *par* p_1*) étant inconnu, il est marqué visité et placé au sommet de la pile. Le même processus s'effectue pour le brin* 0 *(image de* 7 *par* p_2*), et le compteur de brins visités s'incrémente de* 2*. La troisième ligne indique que le dernier brin empilé* 0 *est dépilé et devient le brin courant. Par la suite, le brin* 4 *(image*

de 0 *par* p_1*) étant inconnu, il est marqué visité et placé au sommet de la pile. Le brin* 7 *(image de* 0 *par* p_2*) a déjà été visité donc l'itération de la boucle se termine, et* c *s'incrémente de* 1*. Le processus se poursuit ici jusqu'à ce que tous les brins soient visités, la fonction retournant alors* 1*. Les permutations* p_1 *et* p_2 *forment donc une paire transitive.*

Exemple [16]. *Considérons maintenant les permutations* $p_1 = (0\ 4\ 7\ 2)\ (1\ 5\ 3)\ (6)$ *et* $p_2 = (0)\ (3\ 1\ 6\ 5)\ (2\ 4)\ (7)$ *de taille 8. Après les initialisations, la trace d'exécution de la fonction*

d	S	B: 0	1	2	3	4	5	6	7	c
-	7	0	0	0	0	0	0	0	**1**	1
7	2	0	0	**1**	0	0	0	0	1	2
2	4 0	**1**	0	1	0	**1**	0	0	1	4
4	0	1	0	1	0	1	0	0	1	4
0	vide	1	0	1	0	1	0	0	1	4

TABLE 5.3 – Trace d'exécution de la fonction `b_trans` sur les permutations $p_1 = (0\ 4\ 7\ 2)\ (1\ 5\ 3)\ (6)$ et $p_2 = (0)\ (3\ 1\ 6\ 5)\ (2\ 4)\ (7)$.

`b_trans` sur les deux permutations p_1 *et* p_2 *est décrite dans la table 5.3. La boucle de la fonction stoppe cette fois-ci après* 4 *itérations car la pile* S *est alors vide : seuls 4 brins ont été visités. En effet seuls les brins des cycles* $(0)\ (2\ 4)\ (7)$ *de* p_2 *sont accessibles à partir des brins du cycle* $(0\ 4\ 7\ 2)$ *de* p_1*. Les permutations* p_1 *et* p_2 *ne forment donc pas une paire transitive, et la fonction `b_trans` retourne* 0*.*

Ces structures étant stockées dans des tableaux, on désignera par la suite paire transitive de tableaux toute paire de tableaux (de même longueur) pour lesquels la fonction `b_trans` retourne 1 avec ces deux tableaux en paramètre (dans un ordre quelconque).

Lorsque cet algorithme est appliqué à deux permutations, il caractérise les hypercartes. Il peut s'appliquer également à des permutations particulières (involutions, involutions sans point fixe) pour caractériser des cartes générales ou ordinaires.C'est un composant essentiel des générateurs de cartes et d'hypercartes présentés dans la partie 5.3.

5.3 GÉNÉRATEUR GÉNÉRIQUE DE COUPLES DE TABLEAUX

Les générateurs séquentiels présentés dans le chapitre 3 produisent des structures encodées dans un unique tableau d'entiers. Nous considérons ici des structures encodées par deux tableaux. Pour les (hyper)cartes, il suffit de considérer des couples de permutations. Cependant, dans la perspective de recherches élargies, et pour un coût minimal, il nous a semblé intéressant de créer des générateurs génériques, qui prennent en entrée des couples quelconques de tableaux de même longueur. Ainsi, le générateur PAIR présenté dans le listing 5.9 permet de générer séquentiellement tous les couples de tableaux d'entiers produits par deux générateurs séquentiels de tableaux qu'on lui passe en paramètre.

On parle ici de "couples", c.a.d. de paires ordonnées, car l'ordre des paramètres a une incidence sur la structure construite par ce générateur, contrairement à la fonction `b_trans` qui prend en entrée une paire de tableaux et renvoie le même résultat quel que soit l'ordre des deux tableaux dans la paire.

La fonction `first_pair` génère le premier couple de tableaux. Elle requiert seulement en paramètre, en plus des tableaux a et b et leur taille n, la fonction `first_x` générant le premier tableau a de la famille X, et la fonction `first_y` générant le premier tableau b de la famille Y.

```
1 int first_pair(
2   int a[], int b[], int n,
3   int (*first_x)(int a[], int n),
4   int (*first_y)(int b[], int n)) {
5  return (*first_x)(a,n) ∧  (*first_y)(b,n);
6 }
7
8 int next_pair(
9   int a[], int b[], int n,
10  int (*next_x)(int a[], int n),
11  int (*first_y)(int b[], int n),
12  int (*next_y)(int b[], int n)) {
13  if ((*next_y)(b,n)) return 1;
14  return (*next_x)(a,n) ∧  (*first_y)(b,n);
15 }
```

Listing 5.9 – Générateur générique de couples de tableaux.

La fonction `next_pair` requiert cette fois-ci en paramètre les fonctions de succession `next_x` et `next_y` des générateurs des familles `x` et `y`, ainsi que la fonction `first_y`. Elle génère le couple de tableaux suivant selon l'ordre induit par le processus détaillé ici. Si, à partir des deux tableaux a et b composant le couple courant, il existe un tableau b successeur de la famille Y, alors le couple suivant est constitué du tableau courant a et de ce nouveau tableau b. Sinon, s'il existe un tableau a successeur de la famille X, alors le couple suivant est constitué de ce nouveau tableau a et du premier tableau b de la famille Y. Dans le cas contraire, la fonction `next_x`, et par conséquent la fonction `next_pair`, retourne `0`, signifiant que le dernier couple a été généré. L'obtention de tels couples permet par la suite d'opérer un filtrage par transitivité (ou une autre propriété implémentée sous forme d'une fonction booléenne) pour raffiner les sorties de ce générateur, comme décrit dans la partie 5.4.

5.4 GÉNÉRATEUR GÉNÉRIQUE DE COUPLES TRANSITIFS

Nous présentons ici un générateur générique de couples transitifs de tableaux, par filtrage parmi toutes les couples de tableaux de même taille. Ce générateur, nommé TRANS_PAIR et présenté dans le listing 5.10, est une extension à des couples de tableaux du générateur générique par filtrage pour tableau unique présenté dans le listing 3.7 du chapitre 3.

La fonction `first_trans_pair` génère le premier couple transitif de tableaux des familles X et Y par un filtrage effectué par la fonction `b_trans` présentée dans la partie 5.4. Si le premier couple de tableaux obtenu grâce à un appel à la fonction `first_pair` n'est pas transitif, la fonction `first_trans_pair` génère le couple suivant par un appel à la fonction `next_pair` tant qu'un couple transitif n'est pas généré. Elle requiert donc en paramètre, en plus des tableaux a et b et de leur taille n, les deux fonctions de génération `first_x` et `next_x` des tableaux de la famille X, ainsi que les deux fonctions de génération `first_y` et `next_y` des tableaux de la famille Y. La fonction `next_trans_pair` génère le couple de tableaux suivant par un appel à la fonction `next_pair` tant qu'un couple transitif n'est pas généré. Elle ne requiert donc pas la fonction `first_x` en

paramètre.

```
1  int first_trans_pair(
2    int a[], int b[], int n,
3   int (*first_x)(int a[], int n), int (*next_x)(int a[], int n),
4   int (*first_y)(int b[], int n), int (*next_y)(int b[], int n)) {
5   int tmp = first_pair(a,b,n,*first_x,*first_y);
6   while (tmp ≠ 0 ∧ b_trans(a,b,n) == 0)
7    tmp = next_pair(a,b,n,*next_x,*first_y,*next_y);
8   if (tmp == 0) return 0;
9   return 1;
10 }
11
12 int next_trans_pair(
13   int a[], int b[], int n,
14   int (*next_x)(int a[], int n),
15   int (*first_y)(int b[], int n),
16   int (*next_y)(int b[], int n)) {
17  int tmp;
18  do tmp = next_pair(a,b,n,*next_x,*first_y,*next_y);
19  while (tmp ≠ 0 ∧ b_trans(a,b,n) == 0);
20  if (tmp == 0) return 0;
21  return 1;
22 }
```

Listing 5.10 – Générateur générique de couples transitifs de tableaux.

Ces générateurs génériques seront instanciés en vue de l'obtention de différents générateurs d'(hyper)cartes étiquetées, comme décrit dans la partie 5.5.

5.5 Génération exhaustive de cartes et hypercartes étiquetées

Nous illustrons dans cette partie l'instanciation du générateur générique présenté dans la partie 5.4 pour obtenir des générateurs séquentiels de cartes et hypercartes étiquetées.

Si nous prenons la famille PERM en guise de famille X pour instancier le générateur TRANS_PAIR, nous pouvons instantanément obtenir des générateurs de différents types de cartes suivant la nature de la famille de permutations choisie en guise de famille Y. Ainsi le générateur générique TRANS_PAIR permet d'obtenir par instanciation, d'après les définitions du chapitre 2 :

— un générateur séquentiel d'hypercartes étiquetées si y désigne la famille PERM,
— un générateur séquentiel de cartes générales étiquetées si y désigne la famille INVOL, et
— un générateur séquentiel de cartes ordinaires étiquetées si y désigne la famille EFFI.

Détaillons ce dernier cas. La génération exhaustive de cartes ordinaires étiquetées comportant n arêtes s'obtient par l'appel

```
first_trans_pair(a,b,2*n,(& first_perm),(& next_perm),(& first_-
effi),(& next_effi));,
```

pour générer la première carte, puis par l'exécution de la boucle

```
while(next_trans_pair(a,b,2*n,(& next_perm),(& first_effi),(&
next_effi)) == 1)
```

pour générer séquentiellement les cartes suivantes.

De même, nous avons implémenté un générateur de cartes générales (resp. hypercartes) étiquetées. Pour les cartes générales (resp. hypercartes), nous utilisons le générateur séquentiel efficace d'involutions (resp. de permutations) issu de la bibliothèque `enum` présentée dans le chapitre 3.

5.6 Carte locale

Il existe un grand nombre de cartes étiquetées d'une certaine taille. Certaines sont équivalentes par renommage de leurs brins, ce qui complique leur étude. L'enracinement des cartes permet de supprimer ce problème en les définissant modulo un isomorphisme préservant leur racine, mais cela nécessite le passage au quotient, ce qui est coûteux algorithmiquement. On peut cependant éviter ce problème et restreindre le nombre de cartes étiquetées à générer en fixant l'involution sans point fixe L à l'involution sans point fixe locale (voir la définition 18 de la partie 5.1).

Nous présentons cette nouvelle notion de carte dans la partie 5.6.1. La génération de ces cartes est détaillée dans la partie 5.6.2.

5.6.1 Définition

Définition 19 : Carte locale

Une carte étiquetée (D, R, L) à e arêtes est dite locale si $D = \{0, \ldots, 2e-1\}$ et si L est l'involution sans point fixe locale sur D.

Exemple [17]. *La figure 5.2 présente une carte locale à* 6 *arêtes. Son involution des arêtes est par définition* $L = (0\ 1)\ (2\ 3)\ (4\ 5)\ (6\ 7)\ (8\ 9)\ (10\ 11)$. *Sa permutation des sommets est* $R = (0\ 6\ 5)\ (1\ 2)\ (3\ 4\ 11)\ (7\ 10\ 9\ 8)$.

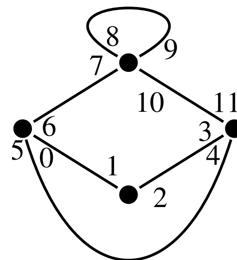


Figure 5.2 – Exemple de carte locale à 6 arêtes.

La conséquence de cette définition est qu'une carte locale peut être définie à partir de sa seule permutation des sommets R, ce qui simplifie sa formalisation et donc son étude. Nous verrons l'intérêt de cette simplification dans les chapitres 8 et 9.

5.6.2 Génération séquentielle de cartes locales

La génération des cartes locales ne consiste pas en une construction de couples comme décrit dans les parties 5.3, 5.4 et 5.5, mais d'un tableau unique encodant la seule per-

mutation R, car l'involution sans point fixe L est fixée. Elle échange $2i$ et $2i+1$ pour tout i appartenant à l'ensemble $\{0,\ldots,n-1\}$, ce qui s'implémente en C sous forme d'une directive

```
#define local(i) (i % 2 == 1 ? i-1 : i+1).
```

La génération exhaustive des cartes locales instancie le générateur générique par filtrage décrit dans la partie 3.4.3 du chapitre 3. Pour cela, on adapte la fonction `b_trans` présentée dans le listing 5.8 en supprimant le paramètre `int p2[]` dans son entête (ligne 1) et en substituant `p2[d]` par `local(d)` dans l'instruction ligne 16 pour obtenir une fonction `b_trans_local`. L'utilisation de cette fonction dans le générateur générique par filtrage se fait grâce aux appels

```
first_filter(a,2*n,(& first_perm),(& next_perm),(& b_trans_-
local));,
```

puis

```
while(next_filter(a,2*n,(& next_perm),(& b_trans_local)) == 1)
```

afin d'obtenir la génération exhaustive séquentielle des cartes locales comportant n arêtes.

5.7 Résultats expérimentaux

Nous présentons dans cette partie des mesures de l'efficacité de notre mode de génération de cartes et hypercartes étiquetées, et de cartes locales.

d	$hm(d)$	$gm(d)$	$om(d)$	$lm(d)$
1	1	1		
2	3	3	2	2
3	26	14		
4	426	138	60	20
5	11 064	1 704		
6	413 640	30 600	8 880	592
7	20 946 960	655 920		
8	1 377 648 720	17 816 400	3 558 240	33 888
9		560 568 960		
10			2 961 826 560	3 134 208
11				
12				423 974 400

TABLE 5.4 – Nombres de cartes de chaque type générées.

La table 5.4 présente les nombres de cartes et d'hypercartes différentes que nous avons générées, en fonction de leur nombre de brins d, indiqué dans la première colonne. La colonne 2 donne le nombre $hm(d)$ d'hypercartes étiquetées à d brins. La colonne 3 donne le nombre $gm(d)$ de cartes générales étiquetées à d brins. Les colonnes 4 et 5 présentent respectivement les nombres $om(d)$ et $lm(d)$ de cartes ordinaires étiquetées et locales.

Nous avons vérifié, conformément aux informations données dans la partie 2.1.4 du chapitre 2, que le nombre d'hypercartes étiquetées générées correspond à la formule

$$hm(d) = (d-1)!A003319(d+1),$$

que le nombre de cartes générales étiquetées générées correspond à la formule

$$gm(d) = (d - 1)!A140456(d)$$

et que le nombre de cartes ordinaires étiquetées générées correspond à la formule

$$om(2e) = (2e - 1)!A000698(e + 1).$$

Le cas des cartes locales à e arêtes est différent, car les brins $2i$ et $2i + 1$ (pour $0 \leq i < e$) y constituent chaque arête. Il existe deux manières de placer les étiquettes des deux brins sur chaque arête (sauf sur l'arête comportant la racine $2e - 1$), et il existe $(e - 1)!$ isomorphismes enracinés renommant chaque arête (l'arête comportant la racine

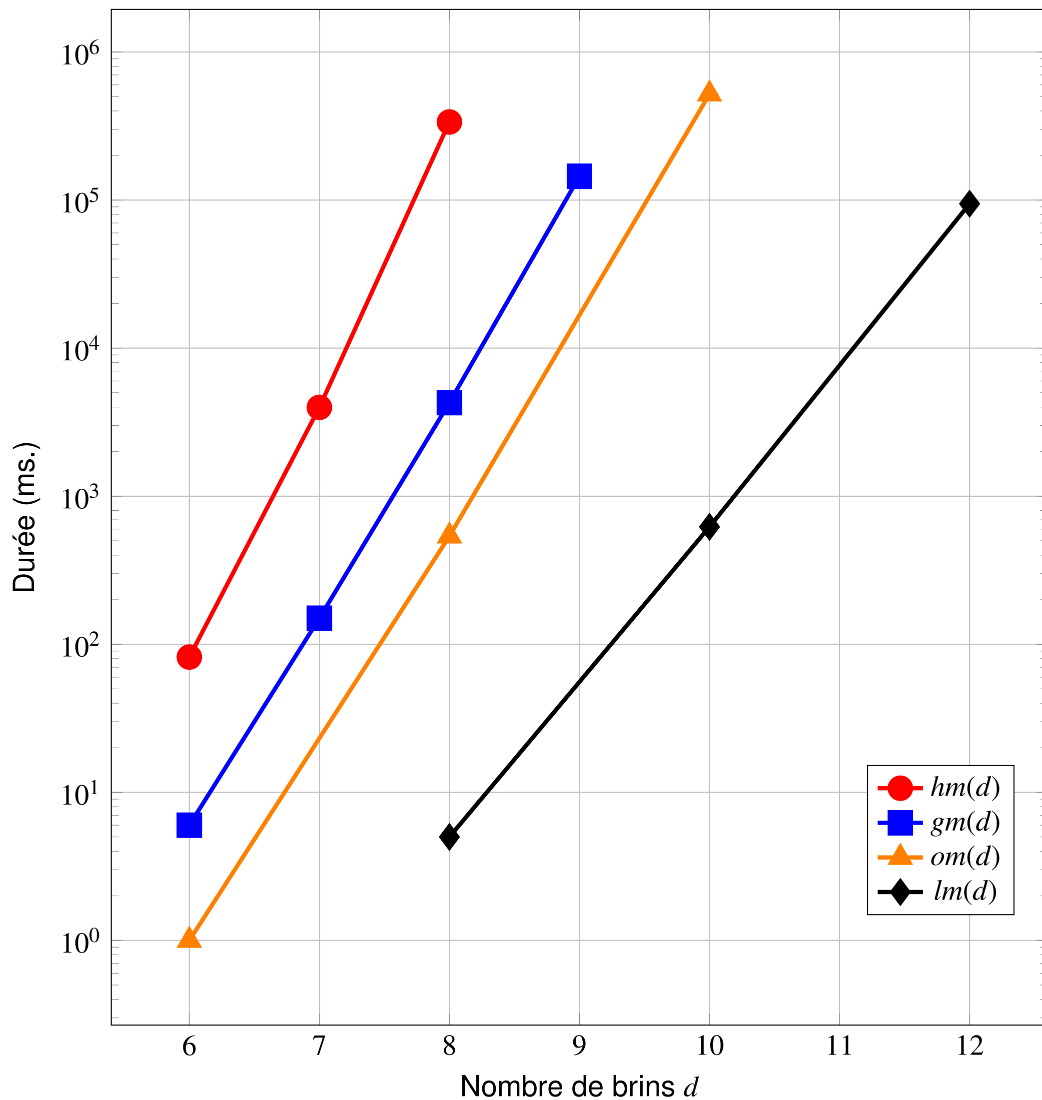


FIGURE 5.3 – Durées de génération d'(hyper)cartes en fonction du nombre de brins.

étant fixée). Ainsi, le nombre de cartes locales à e arêtes est égal au nombre de cartes ordinaires non étiquetées à e arêtes multiplié par $2^{e-1}(e-1)!$, d'où la formule

$$lm(2e) = 2^{e-1}(e-1)!A000698(e+1).$$

La figure 5.3 représente graphiquement les durées de génération pour chaque famille d'(hyper)cartes, en fonction du nombre de brins, sur une échelle logarithmique en ordonnée. Pour les cartes dont le nombre de brins est inférieur à 6, cette durée est infime et n'est pas représentée. Le nombre élevé de cartes et d'hypercartes générées permet de renforcer la confiance dans la correction de nos générateurs. Nous visualisons ici l'un des intérêts des cartes locales qui sont moins nombreuses que les cartes ordinaires et peuvent ainsi être générées dans un temps raisonnable jusqu'à 6 arêtes, au lieu de 5 pour les cartes ordinaires.

5.8 Synthèse et perspectives

Dans ce chapitre, nous avons présenté un générateur efficace original d'involutions sans point fixe, nous avons défini la notion de carte locale, et nous avons implémenté différents générateurs séquentiels de cartes et hypercartes étiquetées.

Une perspective possible serait de mener une étude de complexité algorithmique du générateur séquentiel EFFI.

Une génération exhaustive des cartes non étiquetées, par filtrage parmi les cartes étiquetées, est possible, mais est plus coûteuse algorithmiquement. Une carte non étiquetée est une classe d'équivalence de cartes étiquetées. Pour définir le filtrage, on doit distinguer un représentant unique dans chaque classe d'équivalence, en considérant par exemple la liste des brins obtenus lors d'un parcours en largeur d'abord ou en profondeur d'abord. Cette option n'a pas été menée à son terme, car nous disposons d'autres moyens, plus efficaces, d'obtenir des (hyper)cartes non étiquetées, présentés dans les chapitres 6, 9 et 10.

Conformément aux objectifs fixés au départ, nous abordons maintenant l'étude des cartes non étiquetées, mais d'abord par un moyen détourné. En effet, au lieu de considérer ces cartes à travers leur définition mathématique, nous utilisons une autre manière de les représenter, par des mots. Cette approche est détaillée dans le chapitre 6.

6

Codage des cartes planaires enracinées par des mots

Les générateurs séquentiels de cartes présentés dans le chapitre 5 sont fondés sur des permutations, et énumèrent des cartes étiquetées. Pour l'étude des propriétés des cartes, il suffit souvent d'ignorer leurs étiquettes, c'est-à-dire de considérer leurs classes d'équivalence modulo renommage de leurs étiquettes, appelées cartes enracinées (définition 10 du chapitre 2). Ce sont des cartes non étiquetées. Ce chapitre est le premier consacré à l'énumération des cartes enracinées.

Ce chapitre présente des codages de cartes enracinées par des mots. Nous considérons plus particulièrement ici les cartes planaires enracinées à n arêtes, représentées par des mots à $2n$ lettres, issues d'un alphabet à 4 lettres. Ce codage est un cas particulier d'un codage des cartes enracinées de tout genre qui sort du cadre de ce mémoire. Ce chapitre présente les mots codant les cartes planaires enracinées, ainsi qu'un générateur séquentiel de ces mots. Ce générateur servira notamment dans une étude combinatoire de motifs apparaissant dans ces mots, présentée dans le chapitre 7.

Après un rappel historique sur la représentation des cartes par des mots dans la partie 6.1, la partie 6.2 définit les différentes familles de mots étudiées dans ce chapitre. Ces définitions sont fondées sur la notion de mot de Dyck. La partie 6.3 présente une formalisation des mots de Dyck utile pour concevoir un générateur séquentiel de tableaux représentant ces mots détaillé dans la partie 6.4. Ce dernier sert de base pour implémenter un générateur séquentiel de couples de tels tableaux, puis des mots codant les cartes planaires enracinées, présentés respectivement dans les parties 6.5 et 6.6. La partie 6.7 donne des statistiques de validation par test de ces générateurs.Enfin, la partie 6.8 établit une synthèse de ce chapitre et donne les perspectives de cette étude.

Les fichiers sources des générateurs présentés dans ce chapitre sont disponibles dans les répertoires `generation/height/` et `generation/ncs/` de la bibliothèque `enum`.

6.1 Historique

Des liens entre cartes et mots sont connus depuis la fin des années 60. Nous présentons dans cette partie un bref historique de travaux sur deux familles combinatoires liées aux cartes.

En 1967 R. C. Mullin définit les cartes planaires enracinées avec arbre couvrant distingué,

appelées cartes arborées enracinées (*tree-rooted maps*) [Mullin, 1967]. Il montre que le nombre de cartes arborées enracinées de taille n est C_nC_{n+1}, où

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

désigne le n-ième nombre de Catalan, donné par la séquence A000108 de l'OEIS [The OEIS Foundation Inc., 2010]. Les nombres de Catalan apparaissent dans de multiples domaines de la combinatoire. Ils comptent notamment les arbres binaires et les mots de Dyck de taille n. Cette découverte suggérait donc qu'il pouvait exister des bijections structurelles entre cartes arborées enracinées et certains arbres ou mots. Une bijection entre cartes arborées enracinées et mélanges de mots de Dyck est exposée par A. B. Lehman et T. Walsh dans [Walsh, 1971, Walsh et al., 1972b]. En 1986, R. Cori, S. Dulucq et G. Viennot présentent une bijection entre mélanges de mots de Dyck et certains couples d'arbres [Cori et al., 1986], définie par récurrence sur la longueur du mélange. En 2007, O. Bernardi complète ces résultats avec une bijection directe et non récursive entre cartes arborées enracinées et couples composés d'un arbre et d'une partition non croisée (*non-crossing partition*) [Bernardi, 2007]. Il montre de plus que cette bijection peut se ramener à celle de [Cori et al., 1986] via un codage des cartes arborées enracinées par des mélanges de mots de Dyck.

Dans [Walsh, 1971, Walsh et al., 1972b], A. B. Lehman et T. Walsh abordent également une autre famille combinatoire. Ils exhibent une bijection, cette fois-ci entre cartes planaires enracinées et mélanges de mots de Dyck excluant un certain motif. Plus récemment, A. Giorgetti et V. Senni [Giorgetti et al., 2012] ont introduit des méthodes formelles pour valider des algorithmes générant ces mots.

6.2 Mots codant les cartes planaires enracinées

Les mots constitués de parenthèses bien emboîtées, appelés mots de Dyck, sont présentés dans la partie 6.2.1. Nous définissons ensuite la notion de mélange de mots de Dyck dans la partie 6.2.2. Ils servent de base pour construire des mots codant les cartes planaires enracinées, présentés dans la partie 6.2.3. La bijection avec les cartes planaires enracinées est expliquée dans la partie 6.2.4.

6.2.1 Mots de Dyck

Définition 20 : Mot de Dyck

Un mot de Dyck sur l'alphabet $\{a, \bar{a}\}$ est un mot généré par la grammaire $S \rightarrow \varepsilon \mid aS\bar{a}S$ où ε est le mot vide.

Soit $\mathcal{D}_a$ l'ensemble de tous les mots de Dyck sur l'alphabet $\{a, \bar{a}\}$. Tout mot de Dyck non vide $w \in \mathcal{D}_a$ possède une unique décomposition de la forme $w = a\alpha\bar{a}\beta$, où α et β sont deux mots de Dyck appartenant à $\mathcal{D}_a$ [Deutsch, 1999]. Un mot de Dyck comporte un nombre pair de lettres.

Définition 21 : Longueur et taille d'un mot de Dyck

On appelle longueur d'un mot de Dyck le nombre de ses lettres, et taille sa demi-longueur.

Ces dénominations s'appliqueront à tous les types de mots évoqués dans ce chapitre. Il est établi que les mots de Dyck de taille n sont comptés par les nombres de Catalan.

Exemple [18]. *Il existe cinq mots de Dyck de taille* 3 *:* $a\bar{a}a\bar{a}a\bar{a}$, $a\bar{a}aa\bar{a}\bar{a}$, $aa\bar{a}\bar{a}a\bar{a}$, $aa\bar{a}a\bar{a}\bar{a}$ *et* $aaa\bar{a}\bar{a}\bar{a}$.

On dit qu'une occurrence de la lettre a matche une occurrence de la lettre $\bar{a}$ située à sa droite dans $w \in \mathcal{D}_a$ si le sous-mot de w constitué de toutes les lettres situées entre ces deux occurrences appartient aussi à $\mathcal{D}_a$. Dans ce cas, la paire $(a, \bar{a})$ est appelée un matching dans w. Par exemple, si $w = a\bar{a}aa\bar{a}aa\bar{a}\bar{a}\bar{a}$, alors la seconde occurrence de la lettre a matche la dernière occurrence de la lettre $\bar{a}$ car $a\bar{a}aa\bar{a}\bar{a}$ est un mot de Dyck.

D'autre part, il est établi que la caractérisation des mots de Dyck sous forme de paire de symboles bien emboîtés permet d'obtenir une propriété caractéristique que nous allons abondamment utiliser.

Proposition [2] (Caractérisation des mots de Dyck)**.** *Un mot* w *sur l'alphabet* $\{a, \bar{a}\}$ *appartient à* $\mathcal{D}_a$ *si et seulement s'il vérifie les deux conditions suivantes : (i) le nombre de lettres* a *est égal au nombre de lettres* $\bar{a}$ *dans* w*, et (ii) dans tout préfixe de* w*, le nombre de lettres* a *est supérieur ou égal au nombre de lettres de* $\bar{a}$*.*

6.2.2 Mélange de mots de Dyck

A partir de deux mots de Dyck $v \in \mathcal{D}_a$ et $w \in \mathcal{D}_b$, nous pouvons former un mot s sur l'alphabet $\{a, \bar{a}, b, \bar{b}\}$ en intercalant les lettres de v et w. On obtient alors un shuffle (ou mélange) de deux mots de Dyck.

Définition 22 : Mélange de deux mots de Dyck

Un mélange de (deux) mots de Dyck $v \in \mathcal{D}_a$ et $w \in \mathcal{D}_b$ sur deux alphabets disjoints est un mot s sur l'alphabet $\{a, \bar{a}, b, \bar{b}\}$ dont les restrictions à $\{a, \bar{a}\}$ et $\{b, \bar{b}\}$ sont respectivement v et w.

Soit $\mathcal{S}$ l'ensemble de tous les mélanges de deux mots de Dyck de $\mathcal{D}_a$ et $\mathcal{D}_b$. Par exemple, le mot $s = aab\bar{a}\bar{a}\bar{b}ab\bar{b}\bar{a}$ est un mélange des deux mots de Dyck $aa\bar{a}\bar{a}a\bar{a} \in \mathcal{D}_a$ et $b\bar{b}b\bar{b} \in \mathcal{D}_b$.

Pour tout mélange s, on note $w_a(s)$ (resp. $w_b(s)$) le mot de Dyck de $\mathcal{D}_a$ (resp. $\mathcal{D}_b$) obtenu à partir de s en supprimant les lettres b et $\bar{b}$ (resp. a et $\bar{a}$). La définition 22 peut alors se reformuler ainsi :

Un mot s *est un mélange de* $\mathcal{S}$ *si et seulement si* $w_a(s) \in \mathcal{D}_a$ *et* $w_b(s) \in \mathcal{D}_b$.

Par la suite, nous étendons la définition de $w_a(s)$ et $w_b(s)$ à tout mot dans $\{a, \bar{a}, b, \bar{b}\}^*$. En particulier, si p est un préfixe d'un mélange de mots de Dyck s de $\mathcal{D}_a$ et $\mathcal{D}_b$, $w_a(p)$ (resp. $w_b(p)$) est un préfixe d'un mot de Dyck de $\mathcal{D}_a$ (resp. $\mathcal{D}_b$).

Exemple [19]. *Considérons le mélange* $s = aab\bar{a}\bar{a}\bar{b}ab\bar{b}\bar{a}$*, avec* $w_a(s) = aa\bar{a}\bar{a}a\bar{a}$ *et* $w_b(s) = b\bar{b}b\bar{b}$*. Un préfixe de* s *est par exemple* $p = aab\bar{a}$*. Les mots* $w_a(p) = aa\bar{a}$ *et* $w_b(p) = b$ *sont respectivement des préfixes de* $w_a(s)$ *et* $w_b(s)$*.*

6.2.3 Codage des cartes enracinées

Le codage mis au point par A. B. Lehman et T. Walsh permet de représenter des cartes planaires enracinées (non étiquetées) par des mots sur un alphabet de quatre lettres, initialement $\{(,),[,]\}$. Plus précisément, une carte planaire enracinée (non étiquetée) peut être représentée de façon unique par un mélange de deux mots de Dyck sur les alphabets $\{(,)\}$ et $\{[,]\}$, sans apparition du motif $[(])$ telle que les paires de parenthèses et de crochets composant ce motif soient des matchings (les quatre symboles de ce motif ne sont pas forcément consécutifs dans le mot les contenant).

Le langage de ces mots est appelé “langage de Lehman-Lenormand" par R. Cori [Cori, 1975, page 83]. Dans une communication privée adressée à A. Giorgetti, R. Cori a justifié ce nom comme suit : “Claude Lenormand était un chercheur très inventif du groupe autour de Schützenberger, il rédigeait peu et en avait parlé au cours d'un séminaire”. Nous n'avons pas trouvé d'autres références à cette contribution de C. Lenormand. A. B. Lehman et T. Walsh nomment ces mots “*canonical parenthesis-bracket systems*” [Walsh et al., 1972b], mais ils sont également appelés “*planar Lehman words*” [Giorgetti et al., 2012] et “*Non crossing shuffles*” [Baril et al., 2016] dans la littérature. Nous adoptons dans la suite les dénominations et notations de [Baril et al., 2016], détaillées dans la suite. En particulier les parenthèses et les crochets ouvrants (resp. fermants) de l'alphabet sont remplacés par les lettres a et b (resp. $\bar{a}$ et $\bar{b}$).

Définition 23 : Mélange non croisé

Un mélange s de deux mots de Dyck $v \in \mathcal{D}_a$ et $w \in \mathcal{D}_b$ sera appelé croisé s'il existe un *matching* $(a, \bar{a})$ dans v et un *matching* $(b, \bar{b})$ dans w tels que s s'écrive $s = \alpha b \beta a \gamma \bar{b} \delta \bar{a} \eta$ où $\alpha, \beta, \gamma, \delta$ et η appartiennent à $\{a, \bar{a}, b, \bar{b}\}^*$.
L'occurrence $ba\bar{b}\bar{a}$ sera alors appelée motif croisé.
Un mélange sans motif croisé est appelé mélange non croisé.

Un mélange non croisé sera noté NCS (pour *Non Crossing Shuffle*). On notera alors $\mathcal{NCS} \subset \mathcal{S}$ le sous-ensemble des mélanges non croisés.

Exemple [20]. *Le mélange $a\bar{a}b\bar{b}a\bar{a}b\bar{b}a\bar{a}$ appartient à $\mathcal{NCS}$, tandis que le mélange $a\bar{a}\mathbf{b}b\bar{b}\mathbf{a}\bar{\mathbf{b}}\bar{\mathbf{a}}$ n'appartient pas à $\mathcal{NCS}$ car il contient un motif croisé (en gras).*

La correspondance entre carte planaire enracinée (non étiquetée) et représentation sous forme de NCS est explicitée dans la partie 6.2.4.

6.2.4 Bijection entre mots et cartes

Nous décrivons ici une fonction qui associe bijectivement un NCS de longueur $2n$ à toute carte planaire enracinée à $2n$ brins. La fonction est définie par un parcours de la carte. Elle est illustrée par la figure 6.1, qui donne un exemple de carte planaire enracinée (1) et sa représentation sous forme de NCS (2). Dans l'ordre de parcours de la carte, à partir de sa racine indiquée par une flèche, les brins sont numérotés dans l'ordre croissant, de 1 à $2n$, et une lettre du NCS est associée à chaque brin, comme détaillé dans le paragraphe suivant.

L'image d'une carte par cette fonction est obtenue en appliquant le processus suivant : Partant de la racine de la carte, nous suivons (c.a.d. nous allons du sommet initial au

sommet final du brin) ou traversons (c.a.d. nous effectuons une rotation autour du sommet incident au brin, dans le sens direct) successivement chaque brin de la carte, de manière à atteindre tous les brins. Si le sommet final du brin courant n'a pas encore été examiné, nous suivons ce brin et nous écrivons la lettre a (étapes 1, 2, 3 et 11 de la figure 6.1). Sinon, si l'arête du brin est atteinte pour la première fois, nous écrivons la lettre b et nous traversons ce brin (étapes 4, 5 et 12 de la figure 6.1). Dans les autres cas, le brin opposé du brin courant a déjà été suivi ou traversé. S'il a été suivi (et donc encodé par la lettre a), nous encodons le brin courant par la lettre $\bar{a}$ et nous suivons ce brin courant. C'est le cas dans les étapes 6, 7, 9 et 14 de la figure 6.1. S'il a été traversé (et donc encodé par la lettre b), nous encodons le brin courant par la lettre $\bar{b}$ et nous traversons ce brin courant. C'est le cas dans les étapes 8, 10 et 13 de la figure 6.1.

La figure 6.1 illustre ce processus : partant du brin 1 (racine), nous suivons les brins 1, 2 et 3 car leur sommet final n'a pas encore été examiné, et nous écrivons $a\ a\ a$. Les sommets finaux des brins 4 et 5 ayant déjà été considérés, et les arêtes correspondantes étant atteintes pour la première fois, nous traversons ces brins et nous écrivons $b\ b$. Les brins 6 et 7 étant les brins opposés des brins 3 et 2 déjà suivis, nous écrivons $\bar{a}\ \bar{a}$ et nous suivons ces brins. Le brin 8 étant l'opposé du brin 5 déjà traversé, nous écrivons $\bar{b}$ et nous le traversons. Ce processus se poursuit jusqu'au brin 14 pour clore le parcours de la carte et obtenir le mot $a\ a\ a\ b\ b\ \bar{a}\ \bar{a}\ \bar{b}\ \bar{a}\ \bar{b}\ a\ b\ \bar{b}\ \bar{a}$.

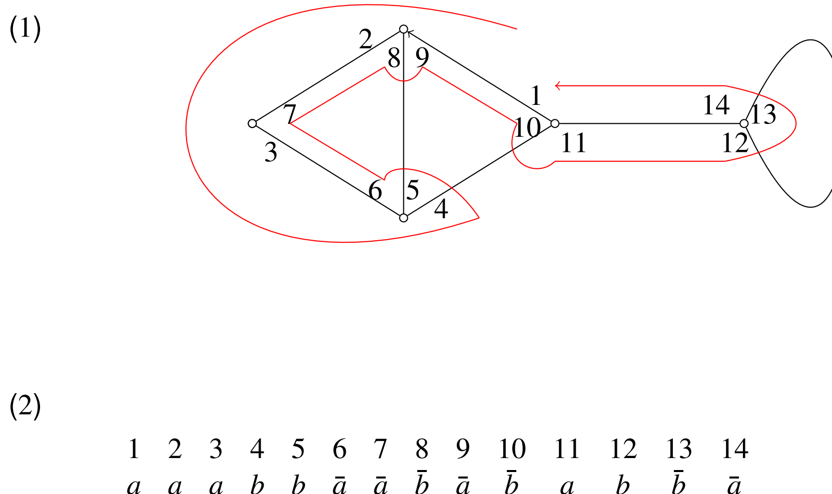


FIGURE 6.1 – Une carte planaire enracinée avec 7 arêtes et sa représentation sous forme de NCS.

Nous abordons dans les parties suivantes la formalisation des différents mots définis dans ce chapitre.

6.3 Formalisation des mots de Dyck

Afin d'obtenir une formalisation des NCS, nous abordons d'abord la formalisation des leurs constituants : les mots de Dyck. Après l'étude des différentes options possibles dans la partie 6.3.1, nous présentons la notion de chemin de Dyck associé à un mot dans la partie 6.3.2, et la représentation d'un mot par son tableau de hauteurs, dans la partie 6.3.3.

6.3.1 Différentes options

Afin d'obtenir un générateur séquentiel de NCS adapté à notre objectif, nous allons suivre le cheminement suivant :

(i) concevoir un générateur séquentiel de mots de Dyck dont on puisse prouver formellement la correction,

(ii) adapter ce générateur pour concevoir un générateur séquentiel de mélanges de mots de Dyck, puis

(iii) adapter ce générateur pour détecter les motifs croisés d'un mélange et obtenir ainsi un générateur séquentiel de NCS.

Cherchons d'abord une formalisation d'un mot de Dyck pour (i), qui facilite la détection de motifs croisés pour (iii).

A partir du contenu de la partie 6.2.1, au moins quatre types de caractérisations des mots de Dyck sur l'alphabet $\{a, \bar{a}\}$ s'offrent à nous pour les formaliser. Détaillons ces différentes options :

1. La proposition 2 repose sur un comptage des lettres a et $\bar{a}$ dans tout préfixe d'un mot w. Les langages de spécification JML et ACSL des langages de programmation Java et C permettent de formaliser cette caractérisation, car ils comportent des quantificateurs numériques \num_of et \sum permettant respectivement de compter et d'additionner les valeurs d'une expression paramétrée par un entier dans un certain intervalle [Engel, 2009]. En encodant la lettre a par $+1$ et la lettre $\bar{a}$ par -1, la caractérisation d'un mot de Dyck w se ramène à établir que la somme des lettres de tout préfixe de w est positive ou nulle, et que la somme de toutes ses lettres est nulle. Cette dernière caractérisation pour un tableau Java w se traduit en JML par la postcondition
```
ensures (\sum int i; 0 <= i && i < w.length; w[i]) == 0;.
```
Malheureusement, les prouveurs actuels ne possèdent pas d'algorithmes suffisamment complets pour effectuer des raisonnements automatiques sur les sommes généralisées, nécessaires pour vérifier la correction de générateurs de mots de Dyck spécifiés selon cette approche. Il est alors nécessaire d'effectuer des preuves interactives, soit par l'ajout manuel dans le système KeY [Beckert et al., 2007] de règles de déduction, soit par l'utilisation d'un assistant de preuve muni d'une bibliothèque gérant ces sommes généralisées, tel que `bigop` en Coq/ssreflect [Bertot et al., 2008]. Ces démarches étant exclues du cadre de cette thèse, nous avons envisagé d'autres options.

2. Les mots de Dyck peuvent être caractérisés par une décomposition inductive, de la même manière que les arbres binaires. Cette caractérisation ne se généralisant pas facilement à un mélange de deux mots de Dyck et ne permettant pas d'y distinguer le motif croisé, nous excluons cette piste.

3. Il est connu que les mots de Dyck sont en bijection avec les fonctions à croissance limitée, présentées dans le chapitre 3. Pour tout mot $w \in \{a, \bar{a}\}^{2n}$ et tout entier naturel j, soit r_j l'entier tel que $w[2j - r_j]$ soit la j-ième lettre a dans w. Alors $w \in \mathcal{D}_a$ si et seulement si $j \mapsto r_j$ est une RGF de taille n. Une RGF permet donc d'obtenir la position des lettres a du mot de Dyck associé. Ce stockage de seulement la moitié des informations permet de représenter facilement les mots de Dyck correspondants, mais la mise en évidence du motif croisé devient en revanche plus complexe. Il est donc plus judicieux d'utiliser une autre représentation.
4. Le nombre de lettres a étant toujours supérieur ou égal au nombre de lettres $\bar{a}$ dans tout préfixe d'un mot $w \in \mathcal{D}_a$, nous pouvons stocker dans un tableau l'excès de a par rapport aux $\bar{a}$. On peut donc caractériser un mot de Dyck par ce tableau appelé tableau des hauteurs, en référence à une représentation graphique classique de ces mots, présentée dans la partie suivante.

6.3.2 Chemins de Dyck

On peut facilement visualiser un mot de Dyck grâce à une représentation graphique appelée chemin de Dyck [Deutsch, 1999]. Nous définissons un tel chemin comme une séquence d'étapes entre deux points consécutifs dans $\mathbb{N} \times \mathbb{N}$. Ainsi, un chemin de Dyck est un chemin $D := s_0, s_1, ..., s_{2n}$ tel que $s_0 = (0,0), s_{2n} = (2n, 0)$ et constitué de deux types d'étapes : Nord-Est (NE) $(s_i = (x,y), s_{i+1} = (x+1, y+1))$ ou Sud-Est (SE) $(s_i = (x,y), s_{i+1} = (x+1, y-1))$. Le nombre d'étapes NE est égal au nombre d'étapes SE, et la longueur du chemin est égale au nombre de ses étapes. Une occurrence d'une lettre a (resp. $\bar{a}$) dans un mot de Dyck w correspond à une étape NE (resp. SE). En outre, en conséquence de la définition d'un mot de Dyck, les coordonnées des points du chemin demeurent positives ou nulles. A titre d'exemple, les cinq mots de Dyck de taille 3 et leur

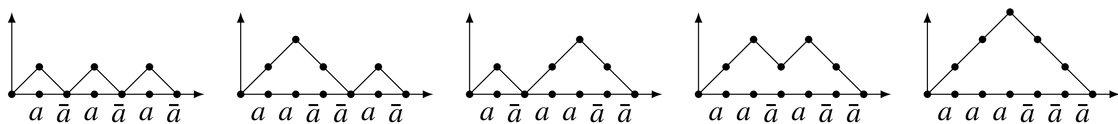


FIGURE 6.2 – Mots de Dyck de taille 3 avec leur chemin de Dyck.

chemin sont donnés dans la figure 6.2.

6.3.3 Tableau de hauteurs

On associe bijectivement à tout mot de Dyck w de longueur $2n$ son tableau de hauteurs h de longueur $2n$ et à valeurs dans $\mathbb{N}$.

Définition 24 : Tableau de hauteurs

Le tableau de hauteurs d'un mot de Dyck w de taille n est le tableau $h[0..2n-1]$ tel que $h[i]$ représente l'excès de lettres a par rapport aux lettres $\bar{a}$ dans un préfixe $w[0..i]$ de w, c.a.d. dans les $i+1$ premières lettres de w.

La figure 6.3 présente le mot de Dyck $aaa\bar{a}aa\bar{a}\bar{a}\bar{a}\bar{a}a\bar{a}$ de taille 6, son chemin et son tableau de hauteurs associés. Notons que la deuxième coordonnée des étapes s_{i+1} du chemin

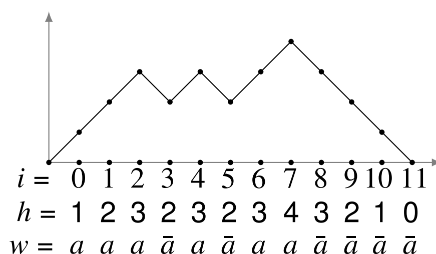


FIGURE 6.3 – Exemple de mot de Dyck de taille 6, chemin w et tableau de hauteurs h associés.

de Dyck est égale à la hauteur $h[i]$ du tableau de hauteurs.

Le tableau de hauteurs d'un mot de Dyck se caractérise de la manière suivante :

Théorème [1]**.** *Le tableau* $h[0..2n-1]$ *est le tableau de hauteurs d'un mot de Dyck* w *de taille* n *si et seulement si*

$$h[0] = 1, \tag{6.1}$$

$$h[2n-1] = 0, \tag{6.2}$$

$$\forall i.\ 0 \le i < 2n \Rightarrow h[i] \ge 0 \tag{6.3}$$

et

$$\forall i.\ 0 \le i < 2n-1 \Rightarrow (h[i+1] = h[i] + 1 \vee h[i+1] = h[i] - 1). \tag{6.4}$$

Démonstration. La première condition de la caractérisation d'un mot de Dyck (proposition 2) stipule que le nombre de lettres a est égal au nombre de lettres $\bar{a}$ dans w, ce qui est équivalent à (6.2).La deuxième condition stipule que dans tout préfixe $w[0..i]$ de w, le nombre de lettres a est supérieur ou égal au nombre de lettres de $\bar{a}$, ce qui est équivalent à (6.3).Cette même condition instanciée avec $i = 0$ équivaut à (6.1). Enfin, un préfixe $w[0..i+1]$ possède une lettre de plus qu'un préfixe $w[0..i]$. L'excès de lettres a par rapport aux lettres $\bar{a}$ dans un préfixe $w[0..i+1]$ diffère donc exactement de 1 par rapport à ce même excès dans $w[0..i]$, ce qui est équivalent à (6.4) et complète cette démonstration. □

Cette formalisation nous permet de concevoir un générateur séquentiel de tableaux de hauteurs, présenté dans la partie 6.4.

6.4 Générateur séquentiel de tableaux de hauteurs

Selon la démarche décrite dans le chapitre 3, nous avons implémenté un générateur séquentiel de tableaux de hauteurs selon l'ordre lexicographique, par révision du suffixe. Cet ordre lexicographique est induit par l'ordre strict $a > \bar{a}$ sur l'alphabet $\{a, \bar{a}\}$ des mots appartenant à D_a. Dans la partie 6.4.1, nous expliquons l'algorithme de révision du suffixe à l'aide des chemins de Dyck correspondants. Nous détaillons sa spécification en ACSL et sa preuve automatique dans la partie 6.4.2.

6.4.1 Révision du suffixe d'un tableau de hauteurs

Ce générateur séquentiel, nommé HEIGHT, est constitué de deux fonctions C `first_height` et `next_height`. C'est une instanciation du patron de génération par révision du suffixe présenté dans le chapitre 3.

Selon l'ordre lexicographique, le plus petit tableau de hauteurs est $(1\ 0)^n$, correspondant au mot de Dyck $(a\bar{a})^n$. Il est généré par la fonction `first_height` donnée dans le listing 6.1.

```
1 void first_height(int h[], int n) {
2  for (int i = 0; i < 2*n; i++) h[i] = 1-i%2;
3 }
```

Listing 6.1 – Fonction `first_height`.

Le processus de révision du suffixe d'un tableau de hauteurs est implémenté par la fonction `next_height` présentée dans le listing 6.2. Il est constitué de plusieurs étapes. Afin de faciliter la vérification automatique de cette fonction, nous l'avons scindée en plusieurs sous-fonctions qui correspondent chacune à une étape élémentaire de la révision du suffixe.

```
1 int next_height(int h[], int n) {
2  int rev,gnd;
3  rev = revision_point(h,n);
4  if (rev == 0) return 0;
5  h[rev] = h[rev]+2;
6  gnd = descent(h,n,rev);
7  zigzag(h,n,gnd);
8  return 1;
9 }
```

Listing 6.2 – Fonction `next_height`.

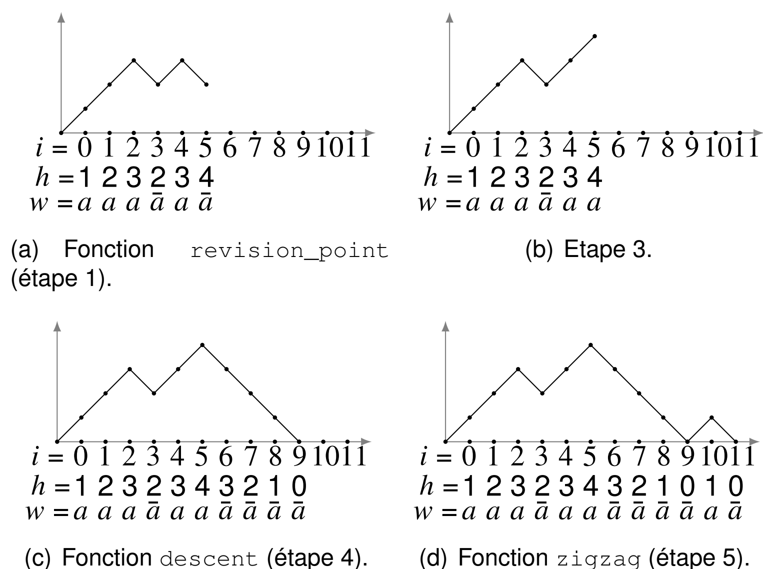


FIGURE 6.4 – Révision du suffixe du tableau de hauteurs présenté dans la figure 6.3, avec son chemin de Dyck associé.

Cette fonction détermine d'abord le point de révision du tableau `h` grâce à un appel de la sous-fonction `revision_point`. Dans le cas où la recherche du point de révision

échoue, le dernier tableau de hauteurs est atteint et cette fonction retourne 0 (ligne 4). Dans le cas contraire, elle augmente la hauteur correspondante de 2 (ligne 5), révise le suffixe du tableau en appelant successivement les sous-fonctions `descent` (ligne 6) et `zigzag` (ligne 7), puis retourne 1.

Détaillons ces différentes étapes à l'aide des chemins de Dyck associés. Ces étapes sont illustrées sur un exemple dans la figure 6.4, à partir du tableau des hauteurs du mot de Dyck w présenté dans la figure 6.3.

1. La recherche du point de révision est implémentée par la sous-fonction `revision_point`, présentée dans le listing 6.3.

```
1 int revision_point(int h[], int n) {
2  int rev = 2*n-2;
3  while (rev ≥ 1 ∧ !(h[rev] == h[rev-1]-1 ∧ h[rev]+2 ≤ 2*n-1-rev)) rev--;
4  return rev;
5 }
```

Listing 6.3 – Fonction `revision_point`.

Cette fonction parcourt le tableau de droite à gauche en cherchant l'abscisse `rev` de la première étape SE rencontrée (condition `h[rev] == h[rev-1]-1`) qui peut être remplacée par une étape NE (condition `h[rev]+2 <= 2*n-1-rev`). Cette dernière condition s'assure qu'on garde la possibilité d'atteindre la hauteur 0 avant la fin du chemin. La fonction `revision_point` retourne ensuite le point de révision `rev`. Par exemple, le point de révision du tableau des hauteurs du mot de Dyck w présenté dans la figure 6.3 est 5. La figure 6.4(a) donne l'état du chemin de Dyck et du tableau de hauteurs de w à l'issue de l'appel de la fonction `revision_point`.

2. Si aucun point de révision n'est trouvé, le dernier tableau de hauteurs est atteint et le processus prend fin (ligne 4 du listing 6.2).

3. Dans le cas contraire, remplacer l'étape SE par une étape NE (ligne 5 du listing 6.2). Cela revient à remplacer une lettre $\bar{a}$ par une lettre a dans le mot de Dyck associé. La figure 6.4(b) donne l'état du chemin de Dyck et du tableau de hauteurs de w à l'issue de cette action.

4. Placer ensuite autant d'étapes SE que nécessaire pour atteindre la hauteur 0. Cette étape est implémentée par la sous-fonction `descent` présentée dans le listing 6.4. Elle inscrit dans le tableau `h` une suite décroissante de hauteurs jusqu'à `0` à partir du successeur de l'indice de révision `rev`. Elle retourne ensuite l'indice de "retour au sol" `rev+h[rev]`, pour lequel la hauteur est 0.

```
1 int descent(int h[], int n, int rev) {
2  for (int k = rev+1; k ≤ rev+h[rev]; k++) h[k] = rev+h[rev]-k;
3  return rev+h[rev];
4 }
```

Listing 6.4 – Fonction `descent`.

Cela revient à placer tous les matchings $\bar{a}$ (s'ils n'étaient pas déjà placés) des lettres a situées dans le préfixe du mot de Dyck associé. La figure 6.4(c) donne l'état du chemin de Dyck et du tableau de hauteurs de w à l'issue de l'appel de la fonction `descent`.

5. Compléter le chemin par des paires (NE,SE) jusqu'à la fin du chemin. Cette étape est implémentée par la sous-fonction `zigzag`, présentée dans le listing 6.5, qui inscrit alternativement `1` puis `0` jusqu'à la fin du tableau `h` à partir du successeur de l'indice de "retour au sol" `gnd`.

```
1 void zigzag(int h[], int n, int gnd) {
2  for (int i = gnd+1; i < 2*n; i++) h[i] = 1-i%2;
3 }
```

Listing 6.5 – Fonction `zigzag`.

Cela revient à placer des paires $a\bar{a}$ jusqu'à la fin du mot de Dyck associé. La figure 6.4(d) donne l'état du chemin de Dyck et du tableau de hauteurs de w à l'issue de l'appel de la fonction `zigzag`.

6.4.2 Spécification ACSL et preuve

Les fonctions de la partie 6.4.1 peuvent être munies d'une spécification ACSL permettant la preuve automatique des propriétés de correction et de progression du générateur séquentiel constitué des deux fonctions `first_height` et `next_height`. Les contrats de ces fonctions utilisent les prédicats ACSL présentés dans le listing 6.6.

```
1 /*@ predicate is_non_neg(int *a, ℤ b) = ∀ ℤ i; 0 ≤ i < b⇒ a[i] ≥ 0;
2   @ predicate is_descent(int *a, ℤ b, ℤ c) = ∀ ℤ i; b+1 ≤ i < c⇒ a[i] == b+a[b]-i;
3   @ predicate is_zigzag(int *a, ℤ b, ℤ c) = ∀ ℤ i; b ≤ i < c⇒ a[i] == 1-i%2;
4   @ predicate is_diff_1(int *a, ℤ b) =
5   @  ∀ ℤ i; 0 ≤ i < b⇒ (a[i+1] == a[i]+1 ∨ a[i+1] == a[i]-1);
6   @ predicate is_height(int *a, ℤ n) =
7   @  a[0] == 1 ∧ a[n-1] == 0 ∧ is_non_neg(a,n) ∧ is_diff_1(a,n-1); */
```

Listing 6.6 – Prédicats ACSL pour les tableaux de hauteurs.

La caractérisation d'un tableau de hauteurs donnée dans le théorème 2 est implémentée par le prédicat `is_height` présenté lignes 6-7. Ce prédicat utilise le prédicat `is_non_neg`, déjà utilisé dans le listing 3.1 du chapitre 3 (ligne 1), ainsi que le prédicat `is_diff_1` (lignes 4-5) stipulant que la différence entre deux hauteurs consécutives est toujours 1. Le prédicat `is_descent` (ligne 2) paraphrase le code de la fonction `descent`, présentée dans le listing 6.4. De même, le prédicat `is_zigzag` (ligne 3) paraphrase le code des fonctions `zigzag` et `first_height`, présentées respectivement dans les listings 6.5 et 6.1. Ces deux prédicats sont utilisés dans des invariants de boucle de ces fonctions afin de prouver certaines de leurs postconditions.

Le premier tableau de hauteurs est construit par la fonction `first_height` présentée avec son contrat ACSL dans le listing 6.7.

```
1 /*@ requires n > 0 ∧ \valid(h+(0..2*n-1));
2   @ assigns h[0..2*n-1];
3   @ ensures is_height(h,2*n); */
4 void first_height(int h[], int n) {
5  /*@ loop invariant 0 ≤ i ≤ 2*n;
6    @ loop invariant is_zigzag(h,0,i);
7    @ loop assigns i, h[0..2*n-1];
8    @ loop variant 2*n-i; */
9  for (int i = 0; i < 2*n; i++) h[i] = 1-i%2;
10 }
```

Listing 6.7 – Fonction `first_height` avec son contrat.

Le contrat ACSL de la fonction `next_height` est présenté dans le listing 6.8. La précondition de ce contrat stipule que le tableau a est de taille positive, est alloué en mémoire, et représente un tableau de hauteurs. La postcondition ligne 3 assure que le tableau a représente un tableau de hauteurs en sortie de fonction. La postcondition ligne 4 assure que si la fonction a modifié le tableau a, celui-ci est supérieur au tableau a en entrée de fonction, selon l'ordre total adopté.

```
1 /*@ requires n > 0 ∧ \valid(h+(0..2*n-1)) ∧ is_height(h,2*n);
2   @ assigns h[0..2*n-1];
3   @ ensures is_height(h,2*n);
4   @ ensures \result == 1⇒ lt_lex{Pre,Post}(h,2*n); */
```

Listing 6.8 – Contrat de la fonction `next_height`.

Ce contrat est prouvé grâce aux différents contrats des sous-fonctions constituant la fonction `next_height`, en suivant le principe de la vérification modulaire, comme pour la vérification par preuve de la fonction `next_effi` présentée dans le chapitre 5. Les postconditions établies dans le contrat d'une sous-fonction sont alors transmises à la sous-fonction suivante en tant que préconditions, permettant ainsi l'établissement de la postcondition de la fonction `next_height` par cascade.

La fonction `revision_point` est munie d'un contrat et d'annotations présentés dans le listing 6.9. Les deux postconditions lignes 4-5 assurent qu'en sortie de fonction le point de révision `rev` vérifie les conditions de sortie de boucle (ligne 11), s'il est différent de `0`. La postcondition ligne 4 assure que nous sommes à une étape NE du chemin de Dyck, et la postcondition ligne 5 assure que le chemin de Dyck garde la possibilité de regagner la hauteur 0 avant la fin du chemin.

```
1 /*@ requires n > 0 ∧ \valid(h+(0..2*n-1)) ∧ is_height(h,2*n);
2   @ assigns \nothing;
3   @ ensures h[0] == 1 ∧ 0 ≤ \result ≤ 2*n-2 ∧ (\result+h[\result])%2 == 1;
4   @ ensures \result ≠ 0⇒ h[\result] == h[\result-1]-1;
5   @ ensures \result ≠ 0⇒ h[\result]+2 ≤ 2*n-1-\result; */
6 int revision_point(int h[], int n) {
7  int rev = 2*n-2;
8  /*@ loop invariant 0 ≤ rev ≤ 2*n-2 ∧ (rev+h[rev])%2 == 1;
9    @ loop assigns rev;
10   @ loop variant rev; */
11 while (rev ≥ 1 ∧ !(h[rev] == h[rev-1]-1 ∧ h[rev]+2 ≤ 2*n-1-rev)) rev--;
12 return rev;
13 }
```

Listing 6.9 – Fonction `revision_point` avec contrat et annotations ACSL.

L'un des points clés de la preuve de correction est d'établir que le point de "retour au sol", décrit dans la partie 6.3.3, est un nombre impair. La troisième partie de la postcondition (ligne 3), qui assure que la somme du point de révision et de la hauteur du tableau à ce point est impaire, permettra de l'établir. Cette partie de la postcondition est prouvée grâce à l'invariant de boucle ligne 8.

```
1 /*@ requires n > 0 ∧ h[0] == 1 ∧ \valid(h+(0..2*n-1)) ∧ 1 ≤ rev ≤ 2*n-2;
2   @ requires h[rev] == h[rev-1]+1 ∧ 0 ≤ h[rev] ≤ 2*n-1-rev;
3   @ requires (rev+h[rev])%2 == 1 ∧ is_diff_1(h,rev) ∧ is_non_neg(h,rev);
4   @ assigns h[rev+1..rev+h[rev]];
5   @ ensures h[0] == 1 ∧ \result%2 == 1 ∧ 1 ≤ \result ≤ 2*n-1 ∧ h[\result] == 0;
6   @ ensures is_non_neg(h,\result) ∧ is_diff_1(h,\result) ∧ \result > rev; */
7 int descent(int h[], int n, int rev) {
8  /*@ loop invariant rev+1 ≤ k ≤ rev+h[rev]+1 ∧ is_descent(h,rev,k);
9    @ loop assigns k, h[rev+1..rev+h[rev]];
10   @ loop variant rev+h[rev]-k; */
11 for (int k = rev+1; k ≤ rev+h[rev]; k++) h[k] = rev+h[rev]-k;
12 return(rev+h[rev]);
13 }
```

Listing 6.10 – Fonction `descent` avec contrat et annotations ACSL.

La fonction `descent` est présentée dans le listing 6.10 avec son contrat et ses annotations ACSL. Son contrat permet d'établir que l'indice de "retour au sol" est impair (deuxième partie de la postcondition ligne 5) et qu'à ce point la hauteur est `0` (quatrième

partie de la postcondition ligne 5). Les contraintes d'un tableau de hauteurs sont ensuite établies jusqu'au point de "retour au sol" (deux premières parties de la postcondition ligne 6). La dernière postcondition, bien que triviale, est nécessaire pour établir la propriété de progression de la fonction `next_height`.

La sous-fonction `zigzag` est présentée dans le listing 6.11 avec son contrat et ses annotations ACSL. Les postconditions prouvées dans la sous-fonction `descent` sont reprises en préconditions qui permettent d'établir la postcondition finale `ensures is_height(h,2*n);` (ligne 4), grâce à l'invariant de boucle ligne 6.

```
1  /*@ requires n > 0 ∧  \valid(h+(0..2*n-1)) ∧  h[0] == 1 ∧  0 ≤ gnd ≤ 2*n-1;
2    @ requires h[gnd] == 0 ∧  gnd%2 == 1 ∧  is_non_neg(h,gnd) ∧  is_diff_1(h,gnd);
3    @ assigns h[gnd+1..2*n-1];
4    @ ensures is_height(h,2*n); */
5  void zigzag(int h[], int n, int gnd) {
6   /*@ loop invariant gnd+1 ≤ i ≤ 2*n ∧  is_zigzag(h,gnd+1,i);
7     @ loop assigns i, h[gnd+1..2*n-1];
8     @ loop variant 2*n-i; */
9   for (int i = gnd+1; i < 2*n; i++) h[i] = 1-i%2;
10 }
```

Listing 6.11 – Fonction `zigzag` avec contrat et annotations ACSL.

Cette postcondition permet de prouver la correction de la fonction `next_height`. Cette fonction étant une instanciation du patron de génération par révision du suffixe présenté dans le chapitre 3, la propriété de progression est également prouvée par WP.

Ce générateur va nous servir de base pour concevoir un générateur séquentiel de couples de tableaux de hauteurs représentant des mélanges de mots de Dyck, présenté dans la partie 6.5.

6.5 Générateur séquentiel de couples de tableaux de hauteurs

D'après la définition 22, un mélange de deux mots de Dyck $w_a \in \mathcal{D}_a$ et $w_b \in \mathcal{D}_b$ est un mot s sur l'alphabet $\{a, \bar{a}, b, \bar{b}\}$ dont les restrictions à $\{a, \bar{a}\}$ et $\{b, \bar{b}\}$ sont respectivement w_a et w_b. Une représentation connue d'un mélange représente le mot w_a par des étapes E (pour a) et O (pour $\bar{a}$) sur l'axe des abscisses, et le mot w_b par des étapes N (pour b) et S (pour $\bar{b}$) sur l'axe des ordonnées) [Bernardi, 2007]. Une telle représentation est donnée

FIGURE 6.5 – Représentation du mélange de mots de Dyck $ab\bar{a}baa\bar{a}\bar{a}b\bar{b}\bar{b}\bar{b}$ par un chemin composé d'étapes N, S, E et O.

dans la figure 6.5 pour le mélange $ab\bar{a}baa\bar{a}\bar{a}b\bar{b}\bar{b}\bar{b}$ de taille 6. Comme nous pouvons le voir, certaines étapes peuvent alors se superposer et la représentation perdre ainsi en lisibilité.

C'est pourquoi nous proposons une représentation d'un mélange par un chemin de Dyck bicolore dont chaque étape porte la couleur a ou b, représentée respectivement en traits pleins et en pointillés. Les restrictions aux traits pleins et aux pointillés de ce chemin de Dyck bicolore constituent respectivement les chemins de Dyck de w_a et w_b. Cette représentation utilise le lemme suivant.

Lemme [1]**.** *Les transformations $b \mapsto a$ et $\bar{b} \mapsto \bar{a}$ transforment tout mélange de $\mathcal{S}$ en un mot de Dyck de $\mathcal{D}_a$.*

Démonstration. Soit s un mélange de $\mathcal{S}$. D'après la proposition 2, (i) dans s, le nombre de lettres a (resp. b) est égal au nombre de lettres $\bar{a}$ (resp. $\bar{b}$), et (ii) dans tout préfixe de s, le nombre de lettres a (resp. b) est supérieur ou égal au nombre de lettres de $\bar{a}$ (resp. $\bar{b}$). Dans le mot s' résultant de l'application des transformations $b \mapsto a$ et $\bar{b} \mapsto \bar{a}$ sur s, si l'on effectue la somme du nombre de lettres a et la somme du nombre de lettres $\bar{a}$ (celles provenant des lettres b et $\bar{b}$ transformées ajoutées à celles initialement présentes dans s), les propriétés (i) et (ii) restent vraies dans $s' = w_a(s')$. Ainsi, $s' \in \mathcal{D}_a$. □

L'implémentation de cette représentation d'un mélange de deux mots de Dyck pourrait être composée d'un unique tableau de hauteurs, et d'un tableau de valeurs booléennes indiquant la couleur de chaque étape. Cependant, en considérant la seule hauteur globale du mélange, nous perdons les hauteurs individuelles de chaque chemin de Dyck. Or, nous devons connaître individuellement les hauteurs correspondant à chaque mot w_a et w_b afin de générer efficacement les mélanges. D'autre part, pour concevoir un générateur de couples de tableaux de hauteurs codant les NCS, nous devons à nouveau connaître individuellement les hauteurs correspondant à chaque mot w_a et w_b afin de mettre en évidence le motif croisé à exclure. C'est pourquoi nous proposons de représenter un mélange par **un couple de tableaux de hauteurs** (h_a, h_b), h_a étant le tableau de hauteurs de w_a, et h_b le tableau de hauteurs de w_b.

Définition 25 : Tableaux de hauteurs d'un mélange

Les tableaux de hauteurs d'un mélange de mots de Dyck w de taille n sont les tableaux $h_a[0..2n-1]$ et $h_b[0..2n-1]$ tels que $h_a[i]$ (resp. $h_b[i]$) représente l'excès de lettres a (resp. b) par rapport aux lettres $\bar{a}$ (resp. $\bar{b}$) dans un préfixe $w[0..i]$ de w, c.a.d. dans les $i+1$ premières lettres de w.

Cette représentation est illustrée par la figure 6.6, qui présente le mélange $ab\bar{a}baa\bar{a}\bar{a}b\bar{b}\bar{b}\bar{b}$ de taille 6 constitué de deux mots de Dyck w_a et w_b, son chemin constitué d'étapes en traits pleins pour w_a et en pointillés pour w_b, ainsi que ses deux tableaux de hauteurs h_a et h_b.

Notons que la deuxième coordonnée des étapes s_{i+1} d'un chemin bicolore d'un mélange de mots de Dyck est égale à la somme des hauteurs $h_a[i]$ et $h_b[i]$. De plus, contrairement aux tableaux de hauteurs présentés dans la partie 6.3.3, les hauteurs peuvent commencer ici à 0 et peuvent rester constantes d'une étape à l'autre.

On a la caractérisation suivante.

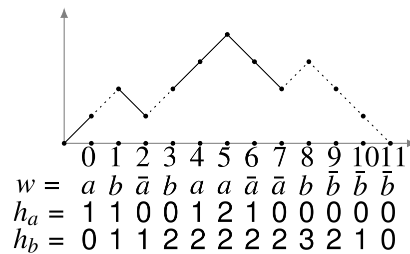


FIGURE 6.6 – Mélange de mots de Dyck avec son chemin bicolore et ses tableaux de hauteurs.

Théorème [2]. *Les tableaux* $h_a[0..2n-1]$ *et* $h_b[0..2n-1]$ *sont les tableaux de hauteurs d'un mélange de mots de Dyck* w *de taille* n *si et seulement si*

$$(h_a[0] = 1 \wedge h_b[0] = 0) \vee (h_a[0] = 0 \wedge h_b[0] = 1), \tag{6.5}$$

$$h_a[2n-1] = 0 \wedge h_b[2n-1] = 0, \tag{6.6}$$

$$\forall i.\ 0 \leq i < 2n \Rightarrow (h_a[i] \geq 0 \wedge h_b[i] \geq 0) \tag{6.7}$$

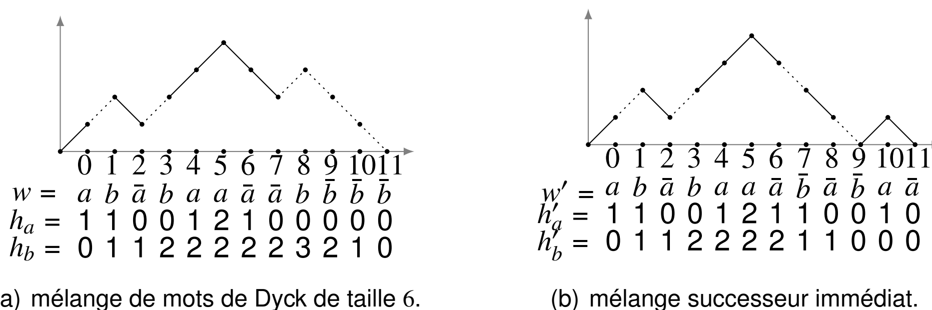
et

$$\begin{aligned}\forall i.\ 0 \leq i < 2n-1 \Rightarrow ((h_a[i+1] = h_a[i] \wedge (h_b[i+1] = h_b[i] + 1 \vee h_b[i+1] = h_b[i] - 1))\\ \vee(h_b[i+1] = h_b[i] \wedge (h_a[i+1] = h_a[i] + 1 \vee h_a[i+1] = h_a[i] - 1))).\end{aligned} \tag{6.8}$$

Démonstration. Ce théorème est une conséquence du théorème 1 et de la définition 25. Les propriétés (6.6) et (6.7) découlent directement des propriétés (6.2) et (6.3). La propriété (6.5) découle de la propriété (6.1) car la première lettre d'un mélange de mots de Dyck peut commencer soit par a, soit par b. La complexité de la propriété (6.8) est liée au fait que les tableaux h_a et h_b comportent ici des paliers. En effet, les hauteurs d'un tableau restent constantes lorsque les hauteurs de l'autre tableau varient. Ainsi, s'il existe un palier dans le tableau h_a, on retrouve conjointement la propriété (6.4) vérifiée par le tableau h_b, et inversement, ce qui établit la propriété (6.8). □

Nous proposons ici un générateur séquentiel qui construit tous les couples de tableaux de hauteurs correspondant aux mélanges de mots de Dyck d'une taille donnée selon l'ordre lexicographique induit par l'ordre $b > a > \bar{b} > \bar{a}$ sur l'alphabet $\{a, \bar{a}, b, \bar{b}\}$. Selon cet ordre, le plus petit mot est $(a\bar{a})^n$, et le plus grand mot est $b^n\bar{b}^n$. Le processus de révision du suffixe implémenté par la fonction de succession de ce générateur agit sur les deux tableaux h_a et h_b.

La figure 6.7(a) présente le mélange $ab\bar{a}baa\bar{a}\bar{a}b\bar{b}\bar{b}\bar{b}$ de taille 6 constitué des deux mots de Dyck $w_a = a\bar{a}aa\bar{a}\bar{a}$ et $w_b = bbb\bar{b}\bar{b}\bar{b}$, son chemin bicolore, ainsi que ses deux tableaux de hauteurs h_a et h_b. La figure 6.7(b) présente le mélange $w' = ab\bar{a}baa\bar{a}\bar{b}\bar{a}\bar{b}a\bar{a}$, successeur du mélange présenté dans la figure 6.7(a) obtenu par application de la fonction de succession, son chemin et ses deux tableaux de hauteurs h'_a et h'_b. Le mélange w a été modifié à partir de l'abscisse 7. Notons que ce mélange n'appartient pas à $\mathcal{NCS}$ car le remplacement de $\bar{a}$ par $\bar{b}$ a créé un motif croisé qui apparaît aux indices 3, 4, 7 et 8.



(a) mélange de mots de Dyck de taille 6. (b) mélange successeur immédiat.

FIGURE 6.7 – Exemple de mélange de mots de Dyck (a) et son successeur immédiat (b) après la révision de son suffixe, avec leurs chemins et leurs tableaux de hauteurs.

La fonction de succession `next_s` présentée dans le listing 6.12 est similaire à celle présentée dans la partie 6.4, mais elle comporte davantage de cas.

Nous allons détailler cet algorithme au niveau des manipulations des lettres composant un mélange, et faire la correspondance avec les manipulations des tableaux h_a et h_b, tout en faisant le lien avec l'implémentation montrée dans le listing 6.12.

Nous suivons les mêmes étapes que pour l'algorithme présenté dans la partie 6.4.1. Cependant, il existe ici plusieurs manières de modifier la lettre située au point de révision, afin d'obtenir le successeur selon l'ordre $b > a > \bar{b} > \bar{a}$. Ainsi, la détermination du point de révision et la modification des hauteurs au point de révision sont rassemblées dans la boucle `for` présentée lignes 3-11. Le principe de détermination du point de révision rev est le suivant : lors d'un parcours de droite à gauche du mélange, trouver la première lettre qui peut être remplacée par une lettre supérieure selon l'ordre $b > a > \bar{b} > \bar{a}$, et la remplacer. On distingue les cas suivants, selon la lettre rencontrée et selon les hauteurs à l'indice courant. Dès que les conditions d'un cas sont remplies, le point de révision est cet indice courant, et la lettre située à cet indice est remplacée par une lettre supérieure. Si aucune de ces conditions n'est vraie, on passe à l'indice précédent.

1. Si l'on rencontre la lettre a, la seule manière d'incrémenter le mot est de la remplacer par une lettre b (ligne 4). Au niveau des tableaux de hauteurs, cela revient à décrémenter $h_a[rev]$ de 1 et incrémenter $h_b[rev]$ de 1.
2. Si l'on rencontre la lettre $\bar{b}$, nous la remplaçons par une lettre a si l'on dispose d'assez de place d'ici la fin du mot pour y placer son matching $\bar{a}$ (ligne 5-6). Cette condition se traduit par $h_a[rev] + h_b[rev] + 2 <= 2n - 1 - rev$ pour les tableaux des hauteurs. Cela revient à incrémenter de 1 à la fois $h_a[rev]$ et $h_b[rev]$.
3. Si l'on rencontre la lettre $\bar{a}$ (ligne 7), deux possibilités s'offrent. Si le mot comporte une lettre b qui n'a pas encore son matching dans le préfixe du mot (condition $h_b[rev] > 0)$, nous remplaçons $\bar{a}$ par ce matching $\bar{b}$ (ligne 8). Cela revient à incrémenter $h_a[rev]$ de 1 et décrémenter $h_b[rev]$ de 1.
4. Dans le cas contraire, on remplace la lettre $\bar{a}$ par une lettre a si l'on dispose d'assez de place d'ici la fin du mot pour y placer son matching $\bar{a}$ (ligne 9). Cela revient à incrémenter $h_a[rev]$ de 2 et garder $h_b[rev]$ égal à $h_b[rev - 1]$. Un remplacement par une lettre b ne donnant pas le plus petit successeur, les différentes possibilités sont épuisées.

Dans l'exemple présenté dans la figure 6.7, le point de révision a pour abscisse 7. En effet les conditions des lignes 4 et 5 ne sont pas remplies, tandis que celle de la ligne 7 l'est

car $h_a[6] = h_a[7] + 1$. De plus $h_b[7] > 0$ donc les instructions de la ligne 8 sont exécutées. La lettre $\bar{a}$ est la première rencontrée à pouvoir être remplacée, ici par la lettre $\bar{b}$ qui est le matching de la lettre b située à l'indice 1.

```
1  int next_s(int ha[], int hb[], int n) {
2
3   for(int rev = 2*n-2; rev > 0; rev--) {
4    if (ha[rev-1] == ha[rev] - 1) { ha[rev]--; hb[rev]++; break; }
5    if (hb[rev-1] == hb[rev] + 1 ∧ ha[rev] + hb[rev] + 2 ≤ 2*n-1-rev) {
6     ha[rev]++; hb[rev]++; break; }
7    if (ha[rev-1] == ha[rev] + 1) {
8     if (hb[rev] > 0) { ha[rev]++; hb[rev]--; break; }
9     if (ha[rev] + hb[rev] + 2 ≤ 2*n-1-rev) { ha[rev] += 2; hb[rev] = hb[rev-1]; break; }
10   }
11  }
12  if (rev == 0)
13   if (ha[0] == 1) { ha[rev]--; hb[rev]++; }
14   else return 0;
15  int k = rev+1;
16  while(k ≤ rev+ha[rev]) { ha[k] = ha[k-1]-1; hb[k] = hb[k-1]; k++; }
17  while(k < rev+hb[rev]+ha[rev]) { ha[k] = 0; hb[k] = hb[k-1]-1; k++; }
18  while(k < 2*n) { ha[k] =  1-k%2; hb[k] = 0; k++; }
19  return 1;
```

Listing 6.12 – Fonction `next_s`.

La fonction `next_s` génère dans un premier temps tous les mélanges commençant par a. Dans le cas où le point de révision est égal à zéro, si le mélange courant commence par la lettre a, elle est remplacé par une lettre b (lignes 12-13). Sinon, le dernier mélange est atteint et la fonction retourne 0 (ligne 14). Si le dernier mélange n'est pas atteint, la fonction retourne 1 après avoir complété le suffixe du successeur immédiat du mélange en trois étapes :

1. Nous plaçons tout d'abord tous les matchings $\bar{a}$ (s'ils n'étaient pas déjà placés) des lettres a situées dans le préfixe du mélange associé (ligne 16). Cela revient à décrémenter jusqu'à 0 (c.a.d. jusqu'à l'indice $rev + h_a[rev] - 1$) les hauteurs du tableau h_a, tandis que sur cet intervalle les hauteurs du tableau h_b demeurent constantes. Dans l'exemple présenté dans le listing 6.12, un matching $\bar{a}$ (de la lettre a située à l'indice 4) est placé à l'indice 8.
2. De même, nous plaçons par la suite tous les matchings $\bar{b}$ (s'ils n'étaient pas déjà placés) des lettres b situées dans le préfixe du mélange associé (ligne 17). Cela revient à décrémenter les hauteurs du tableau h_b jusqu'à 0 (c.a.d. jusqu'à l'indice $rev + h_b[rev] + h_a[rev] - 1$), tandis que sur cet intervalle les hauteurs du tableau h_a demeurent égales à 0. Dans l'exemple présenté dans le listing 6.12, un matching $\bar{b}$ (de la lettre b située à l'indice 1) est placé à l'indice 9.
3. Enfin nous plaçons des couples $a\bar{a}$ jusqu'à la fin du mélange courant (ligne 18). Les hauteurs du tableau h_a prennent donc successivement les valeurs 1 et 0 jusqu'à la fin du tableau, tandis que sur cet intervalle les hauteurs du tableau h_b demeurent égales à 0. Dans l'exemple présenté dans le listing 6.12, un couple $a\bar{a}$ est placé aux indices 10 et 11.

6.6 GÉNÉRATEUR SÉQUENTIEL DE MÉLANGES NON CROISÉS

Cette partie présente un générateur séquentiel des couples de tableaux de hauteurs représentant les NCS, par révision du suffixe. Nous l'avons conçu en adaptant le générateur

présenté dans la partie 6.5. L'adaptation consiste en un ajout, dans la phase de détermination du point de révision, d'une recherche d'un éventuel motif croisé $ba\bar{b}\bar{a}$, pour pouvoir l'éviter.

Cette recherche n'a lieu d'être que lors d'une tentative d'ajout de la lettre $\bar{b}$, troisième lettre du motif croisé, qui détermine l'apparition du motif. Dans ce cas, on étudie la position des deux premières lettres b et a de ce motif (s'il existe) dans le préfixe du mélange courant, pour déterminer si ces lettres b et a sont susceptibles de former un motif croisé avec la lettre $\bar{b}$ que l'on essaie d'insérer à cet endroit. Si b est située avant a, l'apparition de la lettre $\bar{b}$ créera un motif croisé et doit donc être exclue. Une fonction auxiliaire `b_ncs`, présentée dans le listing 6.13, effectue cette recherche et ce test.

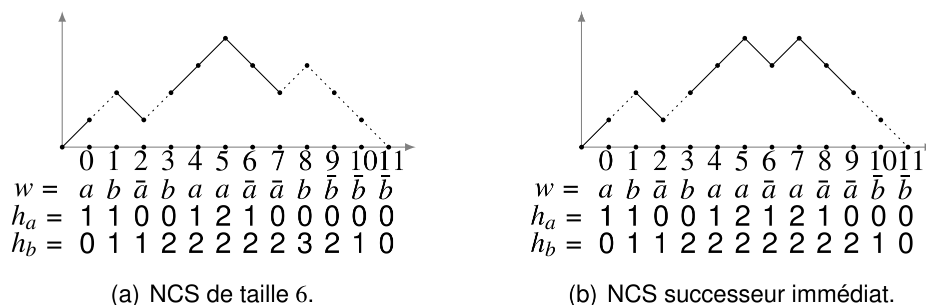
```
1 int b_ncs(int ha[], int hb[], int rev) {
2  int i = rev;
3  do i--;  while (ha[i] > ha[rev] ∧ hb[i] ≥ hb[rev]);
4  return hb[i] < hb[rev];
5 }
```

Listing 6.13 – Fonction `b_ncs`.

Lors d'un parcours du préfixe d'un mélange à partir de l'indice $rev - 1$, on s'arrête dès qu'on rencontre une lettre a ou b qui peut former un motif croisé avec la lettre $\bar{b}$ supposée être à l'indice rev. Cette recherche se traduit par une double condition sur les hauteurs de h_a et h_b (ligne 3). Si, en sortie de boucle, la condition $h_b[i] < h_b[rev]$ est vérifiée, cela signifie que la lettre a est située avant la lettre b et que l'on peut donc placer $\bar{b}$ au point de révision rev. Le suffixe du mot pourra alors être révisé. Dans le cas contraire, $\bar{a}$ ne peut pas être placée dans le mot et on passe au prochain cas de la détermination du point de révision.

Ainsi, la fonction `next_ncs` s'obtient à partir de la fonction `next_s` simplement par l'ajout d'une condition dans la recherche du point de révision. Lors de la détection de la lettre $\bar{a}$ dans la recherche du point de révision du NCS courant, la condition de placement de la lettre $\bar{b}$ (ligne 8 du listing 6.12) devient alors

```
hb[rev] > 0 && b_ncs(ha,hb,rev)).
```



(a) NCS de taille 6. (b) NCS successeur immédiat.

FIGURE 6.8 – Exemple de NCS et son successeur après révision de son suffixe, avec leurs chemins et leurs tableaux de hauteurs.

Illustrons ce processus avec le mélange $ab\bar{a}baa\bar{a}\bar{a}b\bar{b}\bar{b}\bar{b} \in NCS$ qui a servi à illustrer la révision du suffixe d'un mélange (figure 6.7(a)). La figure 6.8 reproduit ce mélange (figure 6.8(a)) et présente son successeur immédiat $ab\bar{a}baa\bar{a}a\bar{a}\bar{a}\bar{b}\bar{b} \in NCS$, son chemin et ses deux tableaux de hauteurs h_a et h_b (figure 6.8(b)). Lors du parcours des tableaux pour la détermination du point de révision, nous rencontrons la lettre $\bar{a}$ à l'indice 7, dont

le matching est situé à l'indice 4. La première lettre b (resp. a) sans son matching rencontrée dans le préfixe du mélange à partir de l'indice 6 est à l'indice 3 (resp. 4), ce qui exclut l'ajout d'une lettre $\bar{b}$ car cela ferait apparaître un motif croisé. La prochaine possibilité consiste à remplacer la lettre courante $\bar{a}$ par une lettre a. La condition ligne 9 est satisfaite ici et la lettre courante $\bar{a}$ est remplacée par la lettre a. On place ensuite aux indices 8 et 9 les matchings $\bar{a}$ des lettres a situées aux indices 7 et 4, et enfin on place aux indices 10 et 11 les matchings $\bar{b}$ des lettres b situées aux indices 3 et 1. Le successeur du NCS de la figure 6.8(a) est alors obtenu sans recours à la dernière étape (ligne 16 du listing 6.12).

Nous sommes assurés que ce successeur est lui-même un NCS car (i) le motif croisé a été évité lors de l'incrémentation du mot au point de révision, et (ii) le fait de placer tous les matchings $\bar{a}$ avant tous les matchings $\bar{b}$ pour terminer la construction du successeur nous assure que ce mélange ne comportera pas de motif croisé et appartiendra donc à $\mathcal{NCS}$.

6.7 Validation des générateurs

Le générateur de tableaux de hauteurs HEIGHT est constitué de 20 lignes de code C. Sa spécification comporte 61 lignes d'ACSL. Lors de sa preuve automatique, 88 conditions de vérifications sont déchargées en 45 secondes.

Nous avons validé l'exhaustivité de ce générateur par une génération exhaustive bornée. Le nombre de tableaux de hauteurs de mots de Dyck de taille n est égal au n-ième nombre de Catalan, donné par la séquence A000108 de l'OEIS [The OEIS Foundation Inc., 2010]. Cette validation a été effectuée jusqu'à la taille $n = 20$ (correspondant à des mots de Dyck de 40 lettres) en 4 minutes 11 secondes, produisant 6 564 120 420 tableaux.

Par manque de temps, nous avons renoncé à effectuer la spécification et la preuve formelle des générateurs de couples de tableaux de hauteurs. Ces générateurs ont cependant été validés par génération exhaustive bornée et comptage.

Les mélanges de taille $n \geq 0$, en bijection avec les cartes arborées enracinées à n arêtes, sont énumérés par la séquence A005568 de l'OEIS [The OEIS Foundation Inc., 2010] donnant le produit de deux nombres de Catalan successifs $C_n C_{n+1}$, dont les premières valeurs sont $1, 2, 10, 70, 588, 5544, 56628, 613470$ pour $n \geq 0$. Le générateur de tableaux de hauteurs représentant les mélanges a été validé par comptage, et par BET en vérifiant, à l'aide d'une fonction C implémentant la caractérisation du couple de tableaux de hauteurs du théorème 2, que tous les mots générés sont des mélanges. La validation complète a été effectuée jusqu'à la taille $n = 10$ (mélanges de 20 lettres) en 3 minutes 33 secondes, produisant 987 369 656 mélanges.

Les NCS de taille $n \geq 0$ sont énumérés par la séquence A000168 de l'OEIS [The OEIS Foundation Inc., 2010], dont les premières valeurs sont $1, 2, 9, 54, 378, 2916, 24057, 208494$ pour $n \geq 0$. Ces mots sont en bijection avec les cartes planaires enracinées (non étiquetées) à n arêtes dont la génération est détaillée dans le chapitre 9. Le générateur de tableaux de hauteurs représentant les NCS a également été validé par comptage et par BET en vérifiant, à l'aide d'une fonction C implémentant la caractérisation du couple de tableaux de hauteurs et d'une autre fonction

C implémentant la mise en évidence d'un motif croisé, que tous les mots générés sont des NCS. La validation complète a été effectuée jusqu'à la taille $n = 11$ (NCS de 22 lettres) en 12 minutes, produisant 1 602 117 468 NCS.

6.8 Synthèse et perspectives

Ce chapitre a présenté trois générateurs séquentiels de mots :
- Un générateur séquentiel formellement vérifié de tableaux de hauteurs codant les mots de Dyck.
- Un générateur séquentiel de couples de tableaux de hauteurs codant les mélanges de mots de Dyck.
- Un générateur séquentiel de couples de tableaux de hauteurs codant les NCS, qui sont en bijection avec les cartes planaires enracinées (non étiquetées).

Une perspective intéressante serait de spécifier le générateur de couples de tableaux de hauteurs de mélanges afin d'effectuer la preuve formelle automatique de la correction de ce générateur.

L'une des applications de ce travail est une étude combinatoire portant sur des motifs apparaissant dans les NCS. Cette étude est présentée dans le chapitre 7.

7

Classes d'équivalence de cartes planaires enracinées

Le chercheur en combinatoire de l'Université de Bourgogne J.-L. Baril a mené plusieurs études combinatoires portant sur les mots de Dyck. Lors d'un échange sur nos travaux respectifs, nous avons jugé prometteur de mêler nos compétences pour étendre l'une de ces études aux mélanges de mots de Dyck et aux NCS, grâce aux générateurs séquentiels présentés dans le chapitre 6. Ce chapitre présente cette étude.

Considérons la relation d'équivalence suivante sur les mots sur un alphabet donné.

Définition 26 : Relation de π-équivalence

Soit A un alphabet fini. Pour tout mot π de A^*, appelé motif, deux mots de A^* de même longueur sont dits π-équivalents si et seulement s'ils contiennent le motif π aux mêmes positions.

Exemple [21]. *Par exemple, si $A = \{a, b, c\}$, le mot $m = bb\mathbf{ba}cca\mathbf{ba}c \in A^*$ est ba-équivalent au mot $m' = cb\mathbf{ba}bcc\mathbf{ba}a \in A^*$ car les occurrences du motif $\mathbf{ba}$ apparaissent aux positions 3 et 8 à la fois dans m et dans m'.* [1]

A partir de cette définition, une question naturelle surgit : si L est un langage sur A (c'est-à-dire une partie de A^*), pour chaque motif $\pi \in A^*$, quel est le nombre de classes de π-équivalence de mots de longueur donnée dans L ?

Dans [Baril et al., 2015a], J.-L. Baril et A. Petrossian abordent ce problème dans le langage des mots de Dyck sur l'alphabet $\{a, \bar{a}\}$. Les auteurs fournissent des fonctions génératrices [2] pour le nombre de classes d'équivalence lorsque les motifs sont de longueur deux : aa, $a\bar{a}$, $\bar{a}\bar{a}$ et $\bar{a}a$. Les mêmes auteurs ont également mené l'étude de cette relation d'équivalence dans le langage des mots de Motzkin sur l'alphabet $\{U, D, F\}$ [Baril et al., 2015b]. Ils fournissent des fonctions génératrices pour le nombre de classes d'équivalence lorsque les motifs sont de longueur un ou deux : U, D, F, UU, DD, DU, UD, UF, DF, FU, FD et FF.

Il nous a semblé intéressant d'entreprendre une étude similaire sur les langages $\mathcal{S}$ et $\mathcal{NCS}$ de mélanges de mots de Dyck sur l'alphabet $\{a, \bar{a}, b, \bar{b}\}$, définis dans le chapitre 6.

1. Par cohérence avec les études antérieures, les indices des lettres dans les mots commencent ici à 1, contrairement aux autres chapitres de ce mémoire où les indices commencent à 0.
2. La notion de fonction génératrice est définie dans la partie 2.1.1.

Les mélanges de $\mathcal{NCS}$ étant en correspondance bijective avec les cartes planaires enracinées, les motifs étudiés dans les mélanges ont des significations concrètes sur les cartes. La table 7 donne les correspondances entre certains motifs de longueur au plus deux et leur signification en termes de carte. Ces correspondances sont basées sur la bijection décrite dans la partie 6.2.4 du chapitre 6. Schématiquement, chaque motif correspond à une petite partie d'une carte rencontrée lors d'un parcours de cette dernière.

Notons que le motif $a\bar{b}$ ne peut pas apparaître dans un mélange représentant une carte planaire enracinée car il crée alors une paire de matchings croisés qui forme le motif croisé exclu dans un NCS.

Motif	a	aa	$a\bar{a}$	b	$b\bar{b}$	$\bar{a}a$	$a\bar{b}$
Carte							N'apparaît pas

TABLE 7.1 – Visualisation de certains motifs d'un mélange sur une carte planaire enracinée.

A partir du générateur séquentiel de NCS présenté dans le chapitre 6, nous avons conçu un programme C qui compte les nombres de classes d'équivalence pour les mélanges de $\mathcal{S}$ et $\mathcal{NCS}$ de petite taille. Ce programme, sa conception et son utilisation seront détaillés dans la partie 7.1. Les premiers nombres de classes d'équivalence calculés expérimentalement par ce programme ont permis d'émettre des conjectures sur des formules générales permettant de les calculer pour toute taille. La suite du travail suit une démarche classique en combinatoire : Nous avons trouvé une grammaire des mots constituant l'ensemble des représentants canoniques des classes d'équivalence, puis nous avons établi une bijection (préservant la longueur) entre cet ensemble et l'ensemble des classes d'équivalence.

La table 7.2 présente l'ensemble des résultats de cette étude. La colonne 1 présente les motifs π étudiés. La colonne 2 donne les premiers nombres a_n de classes de π-équivalence de mots de taille n pour n compris entre 0 et 8. La colonne 3 présente les fonctions génératrices énumérant ces séquences, et la colonne 4 les numéros des séquences OEIS associées (voir la partie 2.1.2 du chapitre 2).

Motif π	$a_n, 0 \leq n \leq 8$	Fonction génératrice	OEIS
$\{a\}, \{\bar{a}\}, \{b\}, \{\bar{b}\}$	$1, 2, 6, 20, 70, 252, 924, 3432, 12870$	$\frac{1}{\sqrt{1-4x}}$	Coefficient binomial central $A000984$
$\{a\bar{a}\}, \{b\bar{b}\}$	$1, 2, 5, 13, 34, 89, 233, 610, 1597$	$\frac{1-x}{1-3x+x^2}$	$A122367$ (Décalage de 1 de $A001519$)
$\{\bar{a}a\}, \{\bar{b}b\}$	$1, 1, 2, 5, 13, 34, 89, 233, 610$	$\frac{1-2x}{1-3x+x^2}$	$A001519$
$\{aa\}, \{\bar{a}\bar{a}\}, \{bb\}, \{\bar{b}\bar{b}\}$	$1, 1, 2, 5, 12, 31, 81, 216, 583$	$\frac{1+x-\sqrt{1-2x-3x^2}}{2x^2+3x-1+\sqrt{1-2x-3x^2}}$	Termes de rangs pairs de $A191385$
$\{ab\}, \{ba\}, \{\bar{a}\bar{b}\}, \{\bar{b}\bar{a}\}$	$1, 1, 2, 4, 9, 20, 47, 109, 262$	$\frac{-2\sqrt{3}\sin\left(1/3\ \arcsin\left(3/2\ \sqrt{3}x\right)\right)}{4\left(\sin\left(1/3\ \arcsin\left(3/2\ \sqrt{3}x\right)\right)\right)^2-3\,x}$	$A138164$

TABLE 7.2 – Enumération des classes de π-equivalence pour les mélanges et les cartes planaires enracinées.

Notons que nous n'avons pas réussi à obtenir le nombre de classes de π-équivalence dans $\mathcal{S}$ et $\mathcal{NCS}$ pour $\pi \in \{\bar{a}b, \bar{b}a, b\bar{a}\}$. Nous laissons donc ces trois cas en problèmes ouverts.

Nous exposons dans la partie 7.1 les aspects expérimentaux qui ont contribué à la découverte des résultats de ce chapitre. Dans la partie 7.2, nous montrons que le problème

de l'énumération des classes de π-équivalence dans $\mathcal{NCS}$ est le même que dans $\mathcal{S}$ si et seulement si π est un motif de longueur au plus deux qui n'appartient pas à l'ensemble $\{\bar{b}a, b\bar{a}, a\bar{b}, \bar{a}b\}$. Dans les parties 7.3, 7.4, 7.5 et 7.6, nous présentons des résultats énumératifs en fournissant des fonctions génératrices pour le nombre de classes de π-équivalence lorsque $\pi \notin \{\bar{b}a, b\bar{a}, a\bar{b}, \bar{a}b\}$. Nous donnons pour finir quelques perspectives dans la partie 7.7.

Les fichiers source de l'étude expérimentale présentée dans ce chapitre sont disponibles dans le répertoire `generation/ncs/` de la bibliothèque `enum`.

7.1 Aspect expérimental

Les résultats présentés dans ce chapitre ont pu dans un premier temps être conjecturés grâce à une approche expérimentale. En effet, nous avons conçu un programme permettant, étant donnés une taille de NCS et un motif à une ou deux lettres, d'obtenir le nombre de classes d'équivalences correspondant.

Le principe d'utilisation de ce programme est le suivant : l'utilisateur exécute le programme en précisant la taille n, qui ne doit pas dépasser 8 (générant des NCS comportant 16 lettres), ce qui correspond à un temps de calcul raisonnable du programme, détaillé plus loin. Il fournit également le motif π étudié, sous forme d'une chaîne de caractères de longueur 1 ou 2. Les couples de tableaux de hauteurs de tous les NCS de taille n sont alors générés à l'aide des fonctions `first_NCS` et `next_NCS`, présentées dans la partie 6.6 du chapitre 6. Pour chaque mot généré, le couple de tableaux de hauteurs est traduit sous forme d'un tableau de caractères encodant le NCS, afin d'y effectuer la recherche du motif voulu.

Les positions (en machine) d'un motif dans un mélange de taille n sont des entiers compris entre 0 et $2n-1$. Nous pouvons donc coder l'ensemble de ces positions par un nombre en base $2n$. Dans un mélange de taille n, on trouve au plus n motifs comportant une ou deux lettres. Ce nombre comportera donc au maximum n chiffres en base $2n$. Si le motif π n'apparaît pas dans le mélange, ce nombre vaut 0. Sinon, si le motif π apparaît k fois dans le mélange ($0 < k \leq n$), ce nombre s'écrira

$$e_{k-1}(2n)^{k-1} + \ldots + e_1(2n)^1 + e_0,$$

les e_i ($0 \leq i \leq k-1$) indiquant les positions du motif, dans l'ordre décroissant pour obtenir les nombres les plus petits possibles. Ce codage est une manière simpliste d'associer un entier différent à chaque vecteur de positions, suffisant ici pour mener notre étude.

Exemple 22. *Dans le mélange $s = bbaa\bar{b}\bar{a}\bar{b}\bar{a}ab\bar{b}\bar{a}$ de taille $n = 6$, les occurrences du motif $\pi = \bar{b}\bar{a}$ apparaissent aux positions $e_0 = 11$, $e_1 = 7$ et $e_2 = 5$. Ces positions sont donc stockées dans le nombre 815 dont la décomposition en base $2n = 12$ est $5\times12^2+7\times12+11$.*

Ces nombres sont inférieurs à $(2n)^n$. En particulier, si $n = 8$, ces nombre sont inférieurs à 16^8, qui correspond au nombre maximal stockable dans un entier de type `unsigned long`.

Pour stocker les positions des occurrences du motif π de tous les mots de taille n (inférieure ou égale à 8), nous utilisons un tableau $stat$ d'entiers de type `unsigned long`, de longueur le nombre $A000168(8) = 1\,876\,446$ de NCS, déclaré en tant que variable globale, et dont tous les éléments sont initialisés à 0.

Muni de ce tableau, le déroulement du programme est le suivant. Le k-ième couple de tableaux de longueur $2n$ généré par le générateur composé des fonctions `first_ncs` et `next_ncs` est décodé pour obtenir le k-ième NCS p de taille n. Etant donné un motif π, la fonction `pos`, présentée dans le listing 7.1, stocke dans $stat[k]$ les positions des occurrences du motif π dans p, et détermine si ce mot appartient à une classe de π-équivalence existante ou non. Elle prend en paramètre un tableau p contenant les caractères du mot, le motif π sous forme d'une chaîne de caractères de une ou deux lettres, l'entier k numérotant le mot étudié, et la taille du mot n.

```
1  int pos(char p[], char *pi, int k, int n) {
2   int i, j = 0, c = 1;
3   stat[k] = 0;
4
5   if (strlen(pi) == 1) {
6    for (i = 2*n-1; i ≥ 0; i--)
7     if (p[i] == pi[0]) { stat[k] += c*(2*n-i); c *= MAXLENGTH; }
8   }
9   else {
10   for (i = 2*n-2; i ≥ 0; i--)
11    if (p[i] == pi[0] ∧ p[i+1] == pi[1]) { stat[k] += c*(2*n-j); c *= MAXLENGTH; }
12  }
13  while (j < k ∧ stat[j] ≠ stat[k]) j++;
14  if (j == k) m++;
15 }
```

Listing 7.1 – Mise en évidence des classes d'équivalence pour un motif donné.

Cette fonction parcourt le mot p de taille n de droite à gauche. Que ce soit pour des motifs de longueur 1 (lignes 5-8) ou 2 (lignes 9-12), si le motif π est trouvé dans le mot à la position i, on modifie le nombre stocké dans $stat[k]$ qui code les positions des occurrences du motif π en base $2n$ (lignes 7 et 11). Si l'élément $stat[k]$ apparaît pour la première fois dans le tableau $stat$, il indique une nouvelle répartition des occurrences du motif dans le mot, mettant ainsi en évidence une nouvelle classe d'équivalence, et le nombre de classes d'équivalence, stocké dans la variable globale m, est incrémenté (lignes 13-14).

Motifs π	Durée de calcul (minutes)
$a, \bar{a}, b, \bar{b}$	35
$a\bar{a}, b\bar{b}$	22
$\bar{a}a, \bar{b}b$	9
$aa, \bar{a}\bar{a}, bb, \bar{b}\bar{b}$	8
$ab, ba, \bar{a}\bar{b}, \bar{b}\bar{a}$	13

TABLE 7.3 – Durées de calcul du nombre de classes de π-équivalence des mélanges pour les tailles de 1 à 8.

Ce programme a permis d'obtenir les nombres présentés dans la table 7.2 jusqu'à $n = 8$ (sauf la valeur pour $n = 0$ qui vaut toujours 1) dans les temps indiqués dans la table 7.3. La première colonne de la table 7.3 comporte les motifs π étudiés. Pour une ligne donnée, le temps de calcul pour chaque motif est approximativement le même. Le temps donné dans la deuxième colonne est la moyenne sur les 2 ou 4 motifs de la première colonne des temps de calcul des nombres de classes de π-équivalence (en secondes) pour des mots de taille inférieure ou égale à 8 (mots de 16 lettres).

Le site de l'OEIS (https://oeis.org) permet d'effectuer une recherche de séquences connues à partir des premiers nombres obtenus expérimentalement. De telles requêtes font parfois apparaître des séquences répertoriées avec un autre numéro, mais qui sont des clones ou des parties d'une certaine séquence. Détaillons les résultats de l'OEIS et leur analyse pour chaque groupe de motifs π.

Une requête sur les premiers nombres $1, 2, 6, 20, 70, 252, 924, 3432, 12870$ de classes de π-équivalence obtenus pour $\pi \in \{a, \bar{a}, b, \bar{b}\}$ donne deux séquences A000984 et A071976, qui diffèrent à partir de $n = 9$. Or, le temps de calcul du nombre de classes d'équivalence pour la taille 9 pour ces motifs devient déraisonnable. Cette limitation n'a cependant pas été un obstacle pour établir les résultats de la partie 7.3 par le raisonnement. Nous avons ensuite constaté que la fonction génératrice du nombre de classes d'équivalence pour ces motifs correspond à la séquence A000984.

Une requête sur les premiers nombres $1, 1, 2, 5, 13, 34, 89, 233, 610$ de classes de π-équivalence obtenus pour $\pi \in \{\bar{a}a, \bar{b}b\}$ donne 2 séquences : A001519 et A011783. Or la deuxième se nomme "Duplicate of A001519". Il n'y a donc aucune ambigüité pour identifier la bonne séquence A001519.

Une requête sur les premiers nombres $1, 2, 5, 13, 34, 89, 233, 610, 1597$ de classes de π-équivalence obtenus pour $\pi \in \{a\bar{a}, b\bar{b}\}$ donne la seule séquence A122367, qui est un décalage vers la droite de 1 de la séquence A001519 comptant le nombre de classes de π-équivalence pour $\pi \in \{\bar{a}a, \bar{b}b\}$.

Une requête sur les premiers nombres $1, 1, 2, 5, 12, 31, 81, 216, 583$ de classes de π-équivalence obtenus pour $\pi \in \{aa, \bar{a}\bar{a}, bb, \bar{b}\bar{b}\}$ ne donne pas une séquence répertoriée mais uniquement les termes de rang pair de la séquence A191385. La séquence A191384 apparaît également, mais c'est une séquence bidimensionnelle T pour laquelle il est précisé que $T(n, 0) = A191385(n)$.

Une requête sur les premiers nombres $1, 1, 2, 4, 9, 20, 47, 109, 262$ de classes de π-équivalence obtenus pour $\pi \in \{ab, ba, \bar{a}\bar{b}, \bar{b}\bar{a}\}$ donne la seule séquence A138164.

La même procédure a été utilisée pour mettre en évidence les premiers nombres de classes d'équivalences des mélanges appartenant à $\mathcal{S}$, en utilisant le générateur séquentiel de mélanges présenté dans la partie 6.5 du chapitre 6. Les positions des occurrences du motif π sont alors stockées dans un tableau $stat$ de longueur $6\,952\,660$, qui est le nombre de mélanges de taille 8 $(= A000108(8) \times A000108(9))$. Nous avons alors pu constater que ces nombres étaient identiques à ceux des mélanges appartenant à $\mathcal{NCS}$. Ce résultat expérimental a entraîné la conjecture du théorème 3. Ce théorème permet d'établir les raisonnements exposés dans les parties suivantes sur des mélanges de $\mathcal{S}$, qui sont des objets plus simples que les mélanges de $\mathcal{NCS}$, et donc de simplifier substantiellement les preuves des théorèmes 5, 6, 7 et 8. Le travail décrit dans les parties suivantes aurait été beaucoup plus difficile sans cette phase expérimentale cruciale.

7.2 Résultats préliminaires

Dans cette partie nous prouvons que, pour certains motifs π, le nombre de classes de π-équivalence dans $\mathcal{S}$ et $\mathcal{NCS}$ sont les mêmes.

Lemme 2. *Si $w = \alpha\bar{a}\beta\gamma$ est un mot de Dyck de $\mathcal{D}_a$ tel que $\beta\gamma$ appartient à $\{a, \bar{a}\}^*$, alors le mot $w' = \alpha\beta\bar{a}\gamma$ appartient également à $\mathcal{D}_a$.*

Démonstration. Le mot de Dyck w' est obtenu à partir de w par un décalage sur la droite de la lettre $\bar{a}$. Le nombre de a et le nombre de $\bar{a}$ restent donc constants dans w'. De plus, dans tout préfixe de w, le nombre de a est supérieur ou égal au nombre de $\bar{a}$ et cette

propriété est également satisfaite dans w'. La caractérisation d'un mot de Dyck donnée dans la proposition 2 du chapitre 6 permet de conclure que w' est dans $\mathcal{D}_a$. □

Lemme 3. *Si $w = \alpha a \beta \bar{a} \gamma$ est un mot de Dyck de $\mathcal{D}_a$ tel que $\beta = \beta_1 \beta_2 \ldots \beta_k$ appartient à $\mathcal{D}_a$, alors, pour tout i tel que $1 \leq i \leq k$, le mot $w' = \alpha a \beta_1 \ldots \beta_{i-1} \bar{a} \beta_i \ldots \beta_k \gamma$ appartient également à $\mathcal{D}_a$.*

Démonstration. Comme w appartient à $\mathcal{D}_a$, (i) le nombre de a et le nombre de $\bar{a}$ sont égaux dans w, et (ii) le nombre de a est supérieur ou égal au nombre de $\bar{a}$ dans tout préfixe. Comme β appartient à $\mathcal{D}_a$, le nombre de a de tout préfixe de $\alpha a \beta_1 \ldots \beta_{i-1}$ est strictement supérieur au nombre de $\bar{a}$. Cela signifie que $\alpha a \beta_1 \ldots \beta_{i-1} \bar{a}$ satisfait la condition (ii). Ainsi, w' satisfait (ii) ce qui implique que w' appartient à $\mathcal{D}_a$. □

Il est évident que ces deux lemmes restent vrais si nous remplaçons la lettre a par b. Nous les utiliserons donc indifféremment pour a ou b.

Théorème 3. *Soit s un mélange de $\mathcal{S}$ et π un motif de longueur au plus deux n'appartenant pas à $\{a\bar{b}, \bar{a}b, \bar{b}a, b\bar{a}\}$. Il existe un mélange s' de $\mathcal{NCS}$ ayant la même longueur que s tel que s et s' sont π-équivalents.*

La preuve de ce théorème s'obtient en exhibant une transformation permettant d'obtenir un mot s' ne contenant pas de motif croisé à partir d'un mot s contenant au moins un motif croisé $ba\bar{b}\bar{a}$. Une transformation possible est l'échange des lettres $\bar{a}$ et $\bar{b}$ dans les occurrences du motif croisé dans le mot, qui le fait disparaître du mot s. Par exemple, le mot $s = b\mathbf{b}a\bar{a}\mathbf{a}a\bar{\mathbf{b}}\bar{a}\bar{b}\bar{\mathbf{a}}$ (dont le motif croisé apparaît en gras) donne le mot $s' = bba\bar{a}aa\bar{a}\bar{a}\bar{b}\bar{b}$ qui ne comporte plus de motif croisé après l'échange $\bar{a} \leftrightarrow \bar{b}$.

Cette transformation ne peut cependant pas s'appliquer si π contient $\bar{a}$ ou $\bar{b}$, puisque les occurrences de π dans s ne seraient pas conservées dans s'. Dans ce cas, une autre transformation possible serait de changer toutes les lettres de s (sauf celles constituant les occurrences du motif π) de manière à ce que la restriction de s' à ces lettres soit un mot de Dyck de la forme $b^\ell \bar{b}^\ell$ pour un certain entier naturel ℓ. Ainsi, si le motif est $\pi = a\bar{a}$ et le mot est par exemple $s = \mathbf{b}a\bar{a}\mathbf{a}ba\bar{a}\bar{b}\bar{\mathbf{b}}\bar{\mathbf{a}}$ (dont le motif croisé apparaît en gras), on obtient par cette transformation le mot $s' = ba\bar{a}bba\bar{a}\bar{b}\bar{b}\bar{b}$ (qui ne contient plus le motif croisé) tel que $w_b(s') = bbb\bar{b}\bar{b}\bar{b}$.

Le même principe est utilisé si le motif est $\pi = \bar{a}a$, mais en remplaçant cette fois-ci toutes les lettres de s (sauf celles constituant les occurrences du motif π) par a ou $\bar{a}$ (car π commence par $\bar{a}$). Par exemple, si le mot est $s = \mathbf{ba}\bar{a}aa\bar{a}a\bar{\mathbf{b}}\bar{a}\bar{\mathbf{a}}$ (dont le motif croisé apparaît en gras), on obtient le mot $s' = aa\bar{a}aa\bar{a}a\bar{a}\bar{a}\bar{a}$, qui ne contient plus le motif croisé et dont la restriction aux lettres ne composant pas le motif π est égale à $aaa\bar{a}\bar{a}\bar{a}$. A partir de ces considérations, nous pouvons maintenant énoncer la preuve complète de ce théorème.

Démonstration. Soit s un mélange de $\mathcal{S}$ n'appartenant pas à $\mathcal{NCS}$, *c.a.d.* contenant une paire de matchings croisés. Le mot s peut alors s'écrire $s = \alpha b \beta a \gamma \bar{b} \delta \bar{a} \eta$ avec $\alpha, \beta, \gamma, \delta$ et η appartenant à $\{a, \bar{a}, b, \bar{b}\}^*$, le motif croisé $ba\bar{b}\bar{a}$ se trouvant le plus à gauche possible dans s.

Nous distinguons ici trois cas : (i) $\pi \in \{a, aa, ab, ba\}$; (ii) $\pi = a\bar{a}$; et (iii) $\pi = \bar{a}a$. Les autres cas sont facilement obtenus par des symétries classiques sur les mélanges ($a \leftrightarrow b$ et miroir).

Cas (i) : Considérons le mot $s' = \alpha b\beta a\gamma\bar{a}\delta\bar{b}\eta$ obtenu à partir de s par un échange de $\bar{a}$ et $\bar{b}$. L'application des lemmes 2 et 3 montrent que s' est aussi un mélange de deux mots de Dyck. Les deux mélanges s et s' appartiennent à la même classe de π-équivalence car cette opération ne peut pas créer ou supprimer l'un des motifs de $\{a, aa, ab, ba\}$. De plus, la paire de matchings croisés la plus à gauche dans s' est décalée à droite par rapport à s. Nous répétons alors ce processus autant de fois que nécessaire pour obtenir un mélange de $\mathcal{NCS}$ ayant la même longueur que s et qui est π-équivalent à s.

Cas (ii) : Soit $s = s_1 s_2 \ldots s_{2k}$ pour $k \geq 1$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Nous obtenons s' à partir de s grâce au processus suivant : nous préservons les positions de toutes les occurrences du motif $a\bar{a}$; les autres lettres de s' sont choisies de telle sorte que la restriction de s' à ces lettres soit un mot de Dyck de la forme $b^\ell \bar{b}^\ell$ pour un certain entier naturel $\ell \geq 0$. On constate alors facilement que $s' \in \mathcal{NCS}$ et que les deux mélanges s et s' appartiennent à la même classe de $a\bar{a}$-équivalence.

Cas (iii) : Soit $s = s_1 s_2 \ldots s_{2k}$ pour $k \geq 1$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Nous obtenons s' à partir de s grâce au processus suivant : nous préservons les positions de toutes les occurrences du motif $\bar{a}a$; les autres lettres de s' sont choisies de telle sorte que la restriction de s' à ces lettres soit un mot de Dyck de la forme $a^\ell \bar{a}^\ell$ pour un certain entier $\ell \geq 0$. On constate alors facilement que $s' \in \mathcal{D}_a \subset \mathcal{NCS}$ et que les deux mélanges s et s' appartiennent à la même classe de $\bar{a}a$-équivalence. □

Les preuves des théorèmes des parties suivantes sont basées sur ce même principe d'obtention d'un mot s' à partir d'un mot s par certaines transformations, qui dépendent du motif π considéré.

Une conséquence immédiate de ce théorème est que le nombre de classes de π-équivalence de $\mathcal{S}$ est égal au nombre de classes de π-équivalence de $\mathcal{NCS}$, quelle que soit la longueur des mots considérés, pour un motif π de longueur au plus deux n'appartenant pas à $\{\bar{b}a, b\bar{a}, a\bar{b}, \bar{a}b\}$. Dans le cadre d'une étude limitée aux motifs de longueur 1 ou 2 n'appartenant pas à $\{\bar{b}a, b\bar{a}, a\bar{b}, \bar{a}b\}$, ce théorème nous permet donc d'étudier les mélanges de $\mathcal{S}$ plutôt que ceux de $\mathcal{NCS}$. Tous les résultats des parties suivantes sont établis pour les deux ensembles $\mathcal{S}$ et $\mathcal{NCS}$.

7.3 EQUIVALENCE MODULO $\pi \in \{a, b, \bar{a}, \bar{b}\}$

Les résultats pour $\pi \in \{b, \bar{a}, \bar{b}\}$ se déduisent de ceux pour $\pi = a$ par des symétries classiques sur les mélanges (miroir et $a \leftrightarrow b$). Ainsi, nous posons $\pi = a$ dans cette partie.

Lemme [4]. *Soit $\mathcal{A}$ le langage des mélanges de $\mathcal{S}$ défini par la grammaire*

$$\mathcal{A} \rightarrow \mathcal{D}_a \mid \mathcal{D}_a b \mathcal{D}_a \bar{b} \mathcal{A},$$

où $\mathcal{D}_a$ est le langage des mots de Dyck sur l'alphabet $\{a, \bar{a}\}$. Il existe une bijection préservant la longueur entre l'ensemble $\mathcal{A}$ et l'ensemble des classes de a-équivalence de $\mathcal{S}$.

Comme pour le théorème 3, on démontre ce lemme en exhibant une transformation permettant d'obtenir un mot $s' \in \mathcal{A}$ à partir de tout mot $s \in \mathcal{S} \backslash \mathcal{A}$. Cette transformation consiste à parcourir le mot s de gauche à droite et à modifier ses lettres une par une (sauf celles

constituant des occurrences du motif π) en plaçant tout d'abord les matchings $\bar{a}$ des lettres a situées dans le préfixe de s modifié, puis la lettre b s'il n'y a pas de matching à compléter dans le préfixe de s modifié.

Par exemple, à partir du mot $s = ba\bar{b}abb\bar{a}\bar{b}\bar{a}\bar{b} \in \mathcal{S}\backslash\mathcal{A}$, on obtient par cette transformation le mot $s' = ba\bar{a}a\bar{a}\bar{b}b\bar{b}b\bar{b} \in \mathcal{A}$.

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ s = & b & a & \bar{b} & a & b & b & \bar{a} & \bar{b} & \bar{a} & \bar{b} \\ s' = & b & a & \bar{a} & a & \bar{a} & \bar{b} & b & \bar{b} & b & \bar{b} \end{array}$$

Le mot s' appartient bien à $\mathcal{A}$ car il est formé d'une concaténation de mots de Dyck de $\mathcal{D}_a$ (éventuellement vides) "englobés" dans des matchings $b\bar{b}$.

Démonstration. Pour $k \geq 1$, soit $s = s_1 s_2 \dots s_{2k}$ un mélange de $\mathcal{S}\backslash\mathcal{A}$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Prouvons qu'il existe un mélange $s' \in \mathcal{A}$ de même longueur que s tel que s et s' appartiennent à la même classe de a-équivalence. Nous définissons le mot s' en appliquant le processus suivant sur s pour i de 1 à $2k$:

- Si $s_i = a$, nous posons $s'_i = a$;

- Sinon, nous distinguons trois cas :
  — (i) Si $w_a(s'_1 \dots s'_{i-1}) \notin \mathcal{D}_a$, nous posons $s'_i = \bar{a}$;
  — (ii) Si $w_a(s'_1 \dots s'_{i-1}) \in \mathcal{D}_a$ et $w_b(s'_1 \dots s'_{i-1}) \notin \mathcal{D}_b$, nous posons $s'_i = \bar{b}$;
  — (iii) Si $w_a(s'_1 \dots s'_{i-1}) \in \mathcal{D}_a$ et $w_b(s'_1 \dots s'_{i-1}) \in \mathcal{D}_b$, nous posons $s'_i = b$.

Notons que $w_a(s'_1 \dots s'_i)$ (resp. $w_b(s'_1 \dots s'_i)$) est un préfixe d'un mot de Dyck de $\mathcal{D}_a$ (resp. $\mathcal{D}_b$) pour tout i, $1 \leq i \leq 2k$. En outre, à la fin du processus, nous avons nécessairement $w_a(s'_1 \dots s'_{2k}) \in \mathcal{D}_a$ et $w_b(s'_1 \dots s'_{2k}) \in \mathcal{D}_b$, ce qui signifie que s' appartient à $\mathcal{S}$.

Puisque le processus préserve les occurrences des lettres a dans s et n'introduit pas d'autres lettres a dans s', s et s' appartiennent à la même classe de a-équivalence. De plus, s' appartient nécessairement à $\mathcal{A}$. En effet ,le processus fixe $s'_i = \bar{a}$ si $w_a(s'_1 \dots s'_{i-1})$ n'est pas un mot de Dyck de $\mathcal{D}_a$; dans tous les autres cas, il fixe $s'_i = b$ ou $s'_i = \bar{b}$ alternativement. Si s' ne contient aucune occurrence de b, alors $s' \in \mathcal{D}_a \subset \mathcal{A}$; sinon, la construction précédente assure que s' a un préfixe de la forme $\alpha b \beta \bar{b}$ avec α et β dans $\mathcal{D}_a$. Cela signifie que s' est engendré par la grammaire $\mathcal{A} \rightarrow \mathcal{D}_a \mid \mathcal{D}_a b \mathcal{D}_a \bar{b} \mathcal{A}$. Ainsi, nous avons $s' \in \mathcal{A}$.

Il reste maintenant à prouver que deux mélanges distincts s et s' de $\mathcal{A}$ ayant la même longueur ne sont pas a-équivalents. Adoptons un raisonnement par l'absurde et supposons que s et s' appartiennent à la même classe.

Nous distinguons trois cas :

- Si $s \in \mathcal{D}_a$ ou $s' \in \mathcal{D}_a$ avec les mêmes positions des occurrences de a, alors nous avons $s = s'$.

- Sinon, s peut être écrit $s = \alpha b \beta \bar{b} \gamma$ avec α et β dans $\mathcal{D}_a$ et γ dans $\mathcal{A}$. De même, s' peut être écrit $s' = \alpha' b \beta' \bar{b} \gamma'$ avec α' et β' dans $\mathcal{D}_a$ et γ' dans $\mathcal{A}$.

Comme α et α' appartiennent à $\mathcal{D}_a$ et ont les mêmes positions des occurrences de la lettre a, nous avons nécessairement $\alpha = \alpha'$. Le même argument appliqué à β implique que $\beta = \beta'$. Nous complétons cette preuve par induction sur la longueur pour γ et γ' dans $\mathcal{A}$ pour conclure que $s = s'$. □

Théorème 4. *La fonction génératrice* $A(x) = \sum_{n\geq 0} a_n x^n$ *du nombre* a_n *de classes de* a*-équivalence de* $\mathcal{S}$ *de taille* n *est donnée par*

$$A(x) = \frac{1}{\sqrt{1-4x}} = \sum_{n\geq 0} \binom{2n}{n} x^n. \tag{7.1}$$

Par conséquent $a_n = \binom{2n}{n}$.

Démonstration. Par le lemme 4, il suffit de trouver une formule close pour la fonction génératrice $A(x)$ du langage $\mathcal{A}$. Tout mélange non vide $s \in \mathcal{A}$ est engendré par la grammaire $\mathcal{A} \to \mathcal{D}_a \mid \mathcal{D}_a b \mathcal{D}_a \bar{b} \mathcal{A}$ où $\mathcal{D}_a$ est l'ensemble des mots de Dyck sur l'alphabet $\{a, \bar{a}\}$. Ceci entraîne l'équation fonctionnelle :

$$A(x) = D(x) + xD(x)^2 A(x)$$

où $D(x) = \frac{1-\sqrt{1-4x}}{2x}$ est la fonction génératrice de Catalan pour l'ensemble $\mathcal{D}_a$, ce qui implique l'égalité 4. □

7.4 Equivalence modulo $\pi \in \{a\bar{a}, b\bar{b}, \bar{a}a, \bar{b}b\}$

Les résultats pour $\pi \in \{b\bar{b}, \bar{b}b\}$ se déduisent de ceux pour $\pi = a\bar{a}$ et $\pi = \bar{a}a$ par des symétries classiques. Ainsi, nous étudions dans cette partie les deux cas $\pi = a\bar{a}$ et $\pi = \bar{a}a$.

Soit $\mathcal{B}$ le sous-ensemble de $\mathcal{S}$ défini par la grammaire $\mathcal{B} \to a\bar{a}\mathcal{B} \mid b\mathcal{B}'\bar{b}\mathcal{B} \mid \varepsilon$ où $\mathcal{B}' \to a\bar{a}\mathcal{B}' \mid \varepsilon$. L'ensemble $\mathcal{B}'$ est à l'évidence constitué de mots de la forme $(a\bar{a})^k$ pour tout $k \geq 0$. En outre, nous constatons facilement que, pour tout $s \in \mathcal{B}$ nous avons $w_b(s) = (b\bar{b})^i$ pour $i \geq 0$ et $w_a(s) = (a\bar{a})^j$ pour $j \geq 0$. Finalement, si $X = a\bar{a}$, alors l'ensemble $\mathcal{B}$ est constitué de tous les mélanges de mots X^i, $i \geq 0$, avec des mots de Dyck de la forme $(b\bar{b})^j$, $j \geq 0$, où les occurrences de $X = a\bar{a}$ ne sont jamais scindées. Par exemple, $ba\bar{a}a\bar{a}\bar{b}ba\bar{a}b\bar{b} \in \mathcal{B}$, tandis que $ba\bar{a}a\bar{b}\bar{a}a\bar{a}b\bar{b} \notin \mathcal{B}$.

Lemme 5. *Il existe une bijection préservant la longueur entre l'ensemble* $\mathcal{B}$ *et l'ensemble des classes de* $a\bar{a}$*-équivalence de* $\mathcal{S}$.

La preuve de ce lemme s'obtient de la même manière que dans la preuve du lemme 4, en exhibant une transformation permettant d'obtenir un mot $s' \in \mathcal{B}$ à partir de tout mot $s \in \mathcal{S}\backslash\mathcal{B}$. Cette transformation consiste à parcourir le mot s de gauche à droite et à modifier ses lettres une par une (sauf celles constituant des occurrences du motif π) afin que les lettres a et $\bar{a}$ contenues dans s' soient uniquement celles constituant les occurrences du motif $\pi = a\bar{a}$. Le mot s' s'obtient cette fois-ci en plaçant tout d'abord les matchings $\bar{b}$ des lettres b situées dans le préfixe de s modifié, puis des paires $b\bar{b}$ s'il n'y a pas de matching à compléter dans ce préfixe.

Par exemple, à partir du mot $s = ba\bar{a}aba\bar{a}\bar{b}\bar{b}\bar{a} \in \mathcal{S}\backslash\mathcal{B}$, on obtient par cette transformation le mot $s' = ba\bar{a}\bar{b}ba\bar{a}\bar{b}b\bar{b} \in \mathcal{B}$.

	1	2	3	4	5	6	7	8	9	10
$s =$	b	a	$\bar{a}$	a	b	a	$\bar{a}$	$\bar{b}$	$\bar{b}$	$\bar{a}$
$s' =$	b	a	$\bar{a}$	$\bar{b}$	b	a	$\bar{a}$	$\bar{b}$	b	$\bar{b}$

Démonstration. Pour $k \geq 1$, soit $s = s_1 s_2 \dots s_{2k}$ un mélange de $\mathcal{S} \backslash \mathcal{B}$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Prouvons qu'il existe un mélange $s' \in \mathcal{B}$ de même longueur que s tel que s et s' appartiennent à la même classe. Nous définissons le mot s' en appliquant le processus suivant à partir de s :

- toutes les occurrences du motif $a\bar{a}$ dans s sont préservées dans s' ;

- pour i de 1 à $2k$ tel que s_i n'appartient pas au motif $a\bar{a}$ dans s, nous distinguons deux cas :

  (i) si $w_b(s'_1 \dots s'_{i-1}) = (b\bar{b})^j$ pour $j \geq 0$, nous posons $s'_i = b$,

  (ii) si $w_b(s'_1 \dots s'_{i-1}) = (b\bar{b})^j b$ pour $j \geq 0$, nous posons $s'_i = \bar{b}$.

Pour tout i ($1 \leq i \leq 2k$), $w_b(s'_1 \dots s'_i)$ peut s'écrire soit $(b\bar{b})^j$, soit $(b\bar{b})^j b$ pour $j \geq 0$. En outre, nous avons $w_a(s'_1 \dots s'_i) = (a\bar{a})^\ell$ pour $\ell \geq 0$ tel que a et $\bar{a}$ apparaissent nécessairement adjacents dans s' dans une occurrence de $a\bar{a}$. Cela signifie que s' est un mélange qui appartient à $\mathcal{B}$. Comme le processus préserve les occurrences du motif $a\bar{a}$, s et s' appartiennent à la même classe de $a\bar{a}$-équivalence.

Moins formellement, s' est obtenu à partir de s en fixant les occurrences de $a\bar{a}$ et en remplaçant toutes les autres lettres par un mot de Dyck de la forme $(b\bar{b})^j$ pour $j \geq 0$. Par conséquent, la définition de $\mathcal{B}$ assure que s' est l'unique élément de $\mathcal{B}$ qui appartient à la même classe de $a\bar{a}$-équivalence que s. □

Théorème [5]. *La fonction génératrice $A(x) = \sum_{n \geq 0} a_n x^n$ du nombre de classes de $a\bar{a}$-équivalence de $\mathcal{S}$ de taille n est donnée par*

$$A(x) = \frac{1 - x}{1 - 3x + x^2} = \sum_{n \geq 0} F_{2n+1} x^n \tag{7.2}$$

où F_m est le m-ième nombre de Fibonacci, (défini par $F_m = F_{m-1} + F_{m-2}$ avec $F_0 = 0$ et $F_1 = 1$). Par conséquent, $a_n = F_{2n+1}$ vérifie la relation de récurrence $a_n = 3a_{n-1} - a_{n-2}$ avec $a_0 = 1$ et $a_1 = 2$.

Démonstration. Grâce au lemme 5, il suffit de fournir la fonction génératrice $B(x)$ de l'ensemble $\mathcal{B}$. Tout mélange $s \in \mathcal{B}$ est obtenu par la grammaire $\mathcal{B} \to a\bar{a}\mathcal{B} \mid b\mathcal{B}'\bar{b}\mathcal{B} \mid \varepsilon$ où $\mathcal{B}' \to a\bar{a}\mathcal{B}' \mid \varepsilon$. Ceci entraîne l'équation fonctionnelle :

$$B(x) = 1 + xB(x) + xB(x)B'(x)$$

où $B'(x) = 1 + xB'(x)$ est la fonction génératrice de l'ensemble $\mathcal{B}'$.

Un simple calcul fournit l'égalité 7.2.

De plus, $F_{2n+1} = F_{2n} + F_{2n-1} = 2F_{2n-1} + F_{2n-2}$ et $F_{2n-2} = F_{2n-1} - F_{2n-3}$ donc $F_{2n+1} = 3F_{2n-1} - F_{2n-3}$, ce qui établit $a_n = 3a_{n-1} - a_{n-2}$. Enfin, $a_0 = F_1 = 1$ et $a_1 = F_3 = 2$. □

Considérons maintenant l'ensemble $\mathcal{B}''$ défini par la grammaire $\mathcal{B}'' = a\mathcal{B}\bar{a} \mid \varepsilon$ où $\mathcal{B}$ est défini au début de cette partie.

Lemme [6]. *Il existe une bijection préservant la longueur entre l'ensemble $\mathcal{B}''$ et l'ensemble des $\bar{a}a$-classes d'équivalence de $\mathcal{S}$.*

La preuve de ce lemme s'obtient avec la même transformation que dans la preuve du lemme 5, en plaçant tout d'abord la lettre a (resp. $\bar{a}$) en tant que première (resp. dernière) lettre de s'. Par exemple, à partir du mot $s = ba\bar{a}ab\bar{a}a\bar{a}\bar{b}\bar{b} \in \mathcal{S}\backslash\mathcal{B}''$, on obtient par cette transformation le mot $s' = ab\bar{a}a\bar{b}\bar{a}ab\bar{b}\bar{a} \in \mathcal{B}''$.

	1	2	3	4	5	6	7	8	9	10
$s =$	b	a	$\bar{a}$	a	b	$\bar{a}$	a	$\bar{a}$	$\bar{b}$	$\bar{b}$
$s' =$	a	b	$\bar{a}$	a	$\bar{b}$	$\bar{a}$	a	b	$\bar{b}$	$\bar{a}$

Démonstration. Soit $s = s_1 s_2 \dots s_{2k}$ ($k \geq 1$) un mélange non vide de $\mathcal{S}\backslash\mathcal{B}''$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Prouvons qu'il existe un mélange $s' \in \mathcal{B}''$ de même longueur que s tel que s et s' appartiennent à la même $\bar{a}a$-classe d'équivalence.

Nous définissons le mélange s' comme suit :

- nous posons $s'_1 = a$ et $s'_{2k} = \bar{a}$;

- toutes les occurrences du motif $\bar{a}a$ dans s sont préservées dans s' ;

- pour i de 2 à $2k-1$ tel que s_i n'appartienne pas à un motif $\bar{a}a$, nous distinguons deux cas :

(i) si $w_b(s'_1 \dots s'_{i-1}) = (b\bar{b})^j$ pour $j \geq 0$, nous posons $s'_i = b$,

(ii) si $w_b(s'_1 \dots s'_{i-1}) = (b\bar{b})^j b$ pour $j \geq 0$, nous posons $s'_i = \bar{b}$.

Pour tout i, $1 \leq i \leq 2k$, $w_b(s'_1 \dots s'_i)$ peut s'écrire soit $(b\bar{b})^j$, soit $(b\bar{b})^j b$ pour $j \geq 0$. En outre, nous avons $w_a(s'_1 \dots s'_i) = (a\bar{a})^\ell$ pour $\ell \geq 0$ de telle manière que a et $\bar{a}$ apparaissent nécessairement adjacents dans s' dans une occurrence $\bar{a}a$, sauf pour $s'_1 = a$ et $s'_{2k} = \bar{a}$. Cela signifie que s' appartient à $\mathcal{B}''$. Comme le processus préserve les occurrences du motif $\bar{a}a$, s et s' appartiennent à la même $\bar{a}a$-classe d'équivalence.

Ce processus assure que s' est l'unique élément de $\mathcal{B}''$ qui appartient à la même $\bar{a}a$-classe d'équivalence que s. □

Théorème 6. *La fonction génératrice* $A(x) = \sum_{n\geq 0} a_n x^n$ *du nombre de classes de* $\bar{a}a$*-équivalence de* $\mathcal{S}$ *de taille* n *est donnée par*

$$A(x) = \frac{1-2x}{1-3x+x^2} = \sum_{n\geq 0} F_{2n-1} x^n \tag{7.3}$$

où F_m *est le* m*-ième nombre de Fibonacci (avec* $F_{-1} = 1$*) Par conséquent,* $a_n = F_{2n-1}$ *vérifie la relation de récurrence* $a_n = 3a_{n-1} - a_{n-2}$ *avec* $a_0 = a_1 = 1$*.*

Démonstration. Grâce au lemme 6, il suffit de fournir la fonction génératrice de l'ensemble $\mathcal{B}'' = a\mathcal{B}\bar{a} \mid \varepsilon$, qui est $B''(x) = xB(x) + 1 = \frac{1-2x}{1-3x+x^2}$. □

7.5 Equivalence modulo $\pi \in \{aa, \bar{a}\bar{a}, bb, \bar{b}\bar{b}\}$

Les résultats pour $\pi \in \{\bar{a}\bar{a}, bb, \bar{b}\bar{b}\}$ sont déduits de ceux pour $\pi = aa$ par des symétries classiques.

Soit C le sous-ensemble de $\mathcal{S}$ défini par la grammaire $C \to C' \mid C'bC'\bar{b}C$ où $C' \to aa\bar{a}C'\bar{a}C' \mid a(C'\backslash\varepsilon)\bar{a}C' \mid \varepsilon$. L'ensemble C' est en fait constitué de mots de Dyck de $\mathcal{D}_a$ tels que toute occurrence de la lettre a est contiguë à une autre occurrence de a.

Lemme 7. *Il existe une bijection préservant la longueur entre l'ensemble C et l'ensemble des classes de aa-équivalence de $\mathcal{S}$.*

La preuve de ce lemme s'obtient avec la même transformation que dans la preuve du lemme 4.

Par exemple, à partir du mot $s = baa\bar{a}\bar{b}aab\bar{a}\bar{a}\bar{a}\bar{b} \in \mathcal{S}\backslash C$, on obtient par cette transformation le mot $s' = baa\bar{a}\bar{a}aa\bar{a}\bar{a}\bar{b}b\bar{b} \in C$.

	1	2	3	4	5	6	7	8	9	10	11	12
$s =$	b	a	a	$\bar{a}$	$\bar{b}$	a	a	b	$\bar{a}$	$\bar{a}$	$\bar{a}$	$\bar{b}$
$s' =$	b	a	a	$\bar{a}$	$\bar{a}$	a	a	$\bar{a}$	$\bar{a}$	$\bar{b}$	b	$\bar{b}$

Démonstration. Soit $s = s_1 s_2 \dots s_{2k}$ $(k \geq 1)$ un mélange de $\mathcal{S}\backslash C$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Montrons qu'il existe un mélange $s' \in C$ de même longueur que s tel que s et s' appartiennent à la même classe d'équivalence.

Nous définissons le mélange s' en appliquant le processus suivant à partir de s :

- toutes les occurrences du motif aa dans s sont préservées dans s' ;

- pour i de 1 à $2k$ tel que s_i n'appartient pas à un motif aa, nous distinguons trois cas :
  (i) Si $w_a(s'_1 s'_2 \dots s'_{i-1}) \in \mathcal{D}_a$ et $w_b(s'_1 s'_2 \dots s'_{i-1}) = (b\bar{b})^j b$ pour $j \geq 0$, nous posons $s'_i = \bar{b}$;
  (ii) Si $w_a(s'_1 s'_2 \dots s'_{i-1}) \in \mathcal{D}_a$ et $w_b(s'_1 s'_2 \dots s'_{i-1}) = (b\bar{b})^j$ pour $j \geq 0$, nous posons $s'_i = b$;
  (iii) Si $w_a(s'_1 s'_2 \dots s'_{i-1}) \notin \mathcal{D}_a$, nous posons $s'_i = \bar{a}$.

Pour tout i, $1 \leq i \leq 2k$, $w_b(s'_1 s'_2 \dots s'_i)$ peut s'écrire soit $(b\bar{b})^j$, soit $(b\bar{b})^j b$ pour $j \geq 0$; $w_a(s'_1 s'_2 \dots s'_i)$ est un préfixe d'un mot de Dyck de $\mathcal{D}_a$, $w_a(s'_1 s'_2 \dots s'_{2k}) \in \mathcal{D}_a$ et $w_b(s'_1 s'_2 \dots s'_{2k}) = (b\bar{b})^j$ pour $j \geq 0$. Ainsi, le mot s' est un mélange de $\mathcal{S}$. En outre, toute occurrence de la lettre a dans s' est toujours contiguë à une autre occurrence de a. Soit $i_1 \geq 1$ (resp. $i_2 > i_1$) la position de la lettre b (resp. $\bar{b}$) la plus à gauche dans s' ; le préfixe $w_a(s'_1 s'_2 \dots s'_{i_1-1})$ (resp. $w_a(s'_1 s'_2 \dots s'_{i_2-1})$) appartient alors à $\mathcal{D}_a$, ce qui implique que le mélange s' est de la forme $\alpha b \beta \bar{b} \gamma$ avec α et β dans C' et γ dans C. Ceci montre que $s' \in C$.

Le processus préserve les positions des occurrences de aa. Ainsi, s et s' appartiennent à la même classe de aa-équivalence.

La preuve de l'unicité de s' dans C est obtenue *mutatis mutandis* à partir de la preuve du lemme 4. □

Théorème 7. *La fonction génératrice $A(x) = \sum_{n \geq 0} a_n x^n$ du nombre a_n de classes de aa-équivalence de $\mathcal{S}$ de taille n est donnée par*

$$A(x) = \frac{1 + x - \sqrt{1 - 2x - 3x^2}}{2x^2 + 3x - 1 + \sqrt{1 - 2x - 3x^2}}. \tag{7.4}$$

Cette séquence correspond aux valeurs de rangs pairs de la séquence A191385 ([The OEIS Foundation Inc., 2010]), dont les premières valeurs pour $n \geq 1$ sont $1, 2, 5, 12, 31, 81, 216, 583, 1590$.

Démonstration. Grâce au lemme 7, il suffit de fournir la fonction génératrice $C(x)$ de l'ensemble C . Tout mélange non vide $s \in C$ est obtenu par la grammaire $C \to C' \mid C'bC'\bar{b}C$ où $C' \to aa\bar{a}C'\bar{a}C' \mid a(C'\backslash\varepsilon)\bar{a}C' \mid \varepsilon$. Soit $C'(x)$ la fonction génératice de C'. A partir de la grammaire ci-dessus, nous déduisons les équations fonctionnelles :

$$C'(x) = x^2C'(x)^2 + x(C'(x) - 1)C'(x) + 1$$

et

$$C(x) = C'(x) + xC'(x)^2C(x)$$

ce qui entraîne $C'(x) = \frac{x+1-\sqrt{1-2x-3x^2}}{2x(x+1)}$ et $C(x) = \frac{1+x-\sqrt{1-2x-3x^2}}{2x^2+3x-1+\sqrt{1-2x-3x^2}}$. □

7.6 EQUIVALENCE MODULO $\pi \in \{ab, ba, \bar{a}\bar{b}, \bar{b}\bar{a}\}$

Les résultats pour $\pi \in \{ba, \bar{a}\bar{b}, \bar{b}\bar{a}\}$ sont déduits de ceux pour $\pi = ab$ par des symétries classiques.

Soit $\mathcal{E}$ l'ensemble des mélanges de $\mathcal{S}$ défini par la grammaire $\mathcal{E} \to \mathcal{E}' \mid \mathcal{E}'b\mathcal{E}'\bar{b}\mathcal{E}$ où $\mathcal{E}' \to ab\mathcal{E}'\bar{b}\mathcal{E}'\bar{a}\mathcal{E}' \mid \varepsilon$. L'ensemble $\mathcal{E}'$ est en fait constitué des mélanges s de $\mathcal{S}$ tels que toute occurrence d'une lettre $x \in \{a, b\}$ apparaît dans un motif ab, et il n'existe pas de matchings $(a, \bar{a})$ et $(b, \bar{b})$ tels que $ab\bar{a}\bar{b}$ apparaisse dans s.

Lemme [8]. *Il existe une bijection préservant la longueur entre l'ensemble $\mathcal{E}$ et l'ensemble des classes de ab-équivalence de $\mathcal{S}$.*

La preuve de ce lemme s'obtient en exhibant une transformation permettant d'obtenir un mot $s' \in \mathcal{E}$ à partir de tout mot $s \in \mathcal{S}\backslash\mathcal{E}$. Cette transformation consiste à parcourir le mot s de gauche à droite et à modifier ses lettres une par une (sauf celles constituant des occurrences du motif π) en plaçant tout d'abord le matching $\bar{a}$ ou $\bar{b}$ de la lettre a ou b sans matching située le plus à droite dans le préfixe de s modifié, puis des paires $b\bar{b}$ s'il n'y a pas de matching à compléter dans ce préfixe.

Par exemple, à partir du mot $s = ab\bar{a}ab\bar{b}\bar{b}\bar{a}a\bar{a} \in \mathcal{S}\backslash\mathcal{E}$, on obtient par cette transformation le mot $s' = ab\bar{b}ab\bar{b}\bar{a}\bar{a}b\bar{b} \in \mathcal{E}$.

	1	2	3	4	5	6	7	8	9	10
$s =$	a	b	$\bar{a}$	a	b	$\bar{b}$	$\bar{b}$	$\bar{a}$	a	$\bar{a}$
$s' =$	a	b	$\bar{b}$	a	b	$\bar{b}$	$\bar{a}$	$\bar{a}$	b	$\bar{b}$

Le mot s' appartient bien à $\mathcal{E}$ car son préfixe $s'[1..8]$ est un mélange de $\mathcal{S}$ tels que toute occurrence d'une lettre $x \in \{a, b\}$ apparaît dans un motif ab, et il n'existe pas de matchings $(a, \bar{a})$ et $(b, \bar{b})$ tels que $ab\bar{a}\bar{b}$ apparaisse dans ce préfixe. Il appartient donc à $\mathcal{E}'$, et sa concaténation avec le matching $b\bar{b}$ forme donc un mot de $\mathcal{E}$.

Démonstration. Soit $s = s_1s_2\ldots s_{2k}$ $(k \geq 1)$ un mélange de $\mathcal{S}\backslash\mathcal{E}$ avec $s_i \in \{a, \bar{a}, b, \bar{b}\}$ pour $1 \leq i \leq 2k$. Prouvons qu'il existe un mélange $s' \in \mathcal{E}$ de même longueur que s tel que s et s' appartiennent à la même classe d'équivalence. Nous définissons le mélange s' en appliquant le processus suivant à partir de s :

- toutes les occurrences du motif ab dans s sont préservées dans s' ;

- pour i de 1 à $2k$ tel que s_i n'appartienne pas à un motif ab, nous distinguons deux cas :

(i) Si $w_b(s'_1 s'_2 \dots s'_{i-1}) \in \mathcal{D}_b$ et $w_a(s'_1 s'_2 \dots s'_{i-1}) \in \mathcal{D}_a$, nous posons $s'_i = b$;
(ii) Si $w_b(s'_1 s'_2 \dots s'_{i-1}) \notin \mathcal{D}_b$ ou $w_a(s'_1 s'_2 \dots s'_{i-1}) \notin \mathcal{D}_a$, alors il existe une lettre $x \in \{a, b\}$ qui n'a pas de matching dans le préfixe $s'_1 s'_2 \dots s'_{i-1}$ (nous choisissons x le plus à droite possible). Si $x = a$, nous posons $s'_i = \bar{a}$, sinon nous posons $s'_i = \bar{b}$.

Montrons maintenant que ce processus produit un mélange de $\mathcal{E}$. Nous distinguons deux cas :

— Supposons que toute lettre $x \in \{a, b\}$ appartienne à une occurrence ab dans s'. Si $s'_i \notin \{a, b\}$, alors $s'_i = \bar{a}$ (resp. $s'_i = \bar{b}$) si la lettre $x \in \{a, b\}$ la plus à droite dans $s'_1 s'_2 \dots s'_{i-1}$ est $x = a$ (resp. $x = b$). Ainsi, $s'_1 s'_2 \dots s'_{i-1} s'_i$ peut être écrit soit $ab\alpha\bar{b}\beta\bar{a}$ soit $ab\alpha\bar{b}$ selon la valeur de s'_i ($\bar{a}$ ou $\bar{b}$), tel que $w_a(\alpha) \in \mathcal{D}_a$, $w_a(\beta) \in \mathcal{D}_a$, $w_b(\alpha) \in \mathcal{D}_b$ et $w_b(\beta) \in \mathcal{D}_b$. En outre, α et β satisfont également la propriété que tout $x \in \{a, b\}$ appartient à une occurrence ab. Ainsi, s' peut être généré par la grammaire $\mathcal{E}' \to ab\mathcal{E}'\bar{b}\mathcal{E}'\bar{a}\mathcal{E}' \mid \varepsilon$.
— A présent, supposons qu'il existe une occurrence b qui n'appartienne pas à un motif ab (le processus assure que toute occurrence de la lettre a appartient à un motif ab dans s'). Nous choisissons la lettre b la plus à gauche ayant cette propriété. Ainsi, s' peut être écrit $s' = \alpha b \beta \bar{b} \gamma$ où $\alpha \in \mathcal{E}'$, $\beta \in \mathcal{E}'$ et $\gamma \in \{a, b, \bar{a}, \bar{b}\}^*$. Par induction, nous avons $\gamma \in \mathcal{E}$, et s' peut être généré par la grammaire $\mathcal{E} \to \mathcal{E}' b \mathcal{E}' \bar{b} \mathcal{E}$.

En considérant les deux cas, s' peut être généré par la grammaire $\mathcal{E} \to \mathcal{E}' \mid \mathcal{E}' b \mathcal{E}' \bar{b} \mathcal{E}$ où $\mathcal{E}' \to ab\mathcal{E}'\bar{b}\mathcal{E}'\bar{a}\mathcal{E}' \mid \varepsilon$.

Il est clair que deux mélanges s et s' de $\mathcal{E}'$ appartenant à la même classe de ab-équivalence sont nécessairement identiques. Par un argument récursif, cela entraîne l'unicité d'un mélange dans $\mathcal{E}$ pour une classe de ab-équivalence donnée. □

Théorème 8. *La fonction génératrice* $A(x) = \sum_{n \geq 0} a_n x^n$ *du nombre* a_n *de classes de* ab*-équivalence de* $\mathcal{S}$ *de taille* n *est donnée par*

$$A(x) = \frac{-2\sqrt{3}\sin\left(1/3 \arcsin\left(3/2\sqrt{3}x\right)\right)}{4\left(\sin\left(1/3 \arcsin\left(3/2\sqrt{3}x\right)\right)\right)^2 - 3x}. \tag{7.5}$$

C'est la séquence A138164 de [The OEIS Foundation Inc., 2010], car la fonction génératrice $A(x)$ est égale à la fonction génératrice $1/(1 - v - v^2)$ avec $v = (2/\sqrt{3}) sin(arcsin(3x\sqrt{3}/2)/3)$ fournie dans la page de la séquence A138164 de l'OEIS. Cette page fournit également une formule de calcul des a_n. Les premières valeurs pour $n \geq 1$ sont $1, 2, 4, 9, 20, 47, 109, 262, 622$.

Démonstration. Grâce au lemme 8, il suffit de fournir la fonction génératrice $E(x)$ de l'ensemble $\mathcal{E}$. Tout mélange non vide $s \in \mathcal{E}$ est obtenu par la grammaire $\mathcal{E} \to \mathcal{E}' \mid \mathcal{E}' b \mathcal{E}' \bar{b} \mathcal{E}$ où $\mathcal{E}' \to ab\mathcal{E}'\bar{b}\mathcal{E}'\bar{a}\mathcal{E}' \mid \varepsilon$.

Soit $E'(x)$ la fonction génératrice de l'ensemble $\mathcal{E}'$. A partir de la grammaire ci-dessus, nous déduisons les équations fonctionnelles

$$E'(x) = 1 + x^2 E'(x)^3$$

et

$$E(x) = E'(x) + xE'(x)^2 E(x).$$

La fonction génératrice $E'(x)$ est connue (voir A001764 de [The OEIS Foundation Inc., 2010]) et donnée par

$$E'(x) = \frac{2\sqrt{3}\sin\left(1/3 \ \arcsin\left(3/2 \ \sqrt{3}x\right)\right)}{3x}.$$

Un simple calcul donne

$$E(x) = \frac{-2\sqrt{3}\sin\left(1/3 \ \arcsin\left(3/2 \ \sqrt{3}x\right)\right)}{4\left(\sin\left(1/3 \ \arcsin\left(3/2 \ \sqrt{3}x\right)\right)\right)^2 - 3\,x}.$$

□

Notons que la fonction génératrice E est égale à la fonction génératrice

$1/(1 - v - v^2)$ avec $v = (2/\sqrt{3})sin(arcsin(3x\sqrt{3}/2)/3)$

donnée dans la page OEIS de la séquence A138164.

7.7 Synthèse et perspectives

Nous avons présenté une énumération des classes de π-équivalence de cartes planaires enracinées représentées par des mélanges de mots de Dyck, pour un motif π de longueur au plus deux n'appartenant pas à $\{a\bar{b}, \bar{a}b, b\bar{a}, \bar{b}a\}$. Cette étude combinatoire (excepté l'aspect expérimental détaillé dans la partie 7.1) a fait l'objet d'une publication dans la revue *Discrete Mathematics* [Baril et al., 2016].

Les perspectives de ce travail sont nombreuses, en lien ou pas avec les cartes. La même étude pour les trois motifs restants de longueur 2 ($\bar{b}a, b\bar{a}$ et $\bar{a}b$) est une première perspective de recherche intéressante. L'étude du nombre de classes de π-équivalence de NCS pourrait également être faite pour des motifs plus longs. Il est aussi possible d'envisager l'étude d'autres mots, comme par exemple des mélanges de mots de Dyck et de mots de Motzkin, etc.

Toujours dans l'optique de générer les cartes non étiquetées, nous allons maintenant nous orienter vers un autre mode de génération des cartes, qui va nous permettre de mettre en œuvre différentes techniques de test et de preuve. Cette étude fait l'objet du chapitre 8.

8

Méthodologie : Test et preuve pour les cartes

Nous avons présenté dans le chapitre 5 un générateur séquentiel de cartes combinatoires étiquetées (d'après leur définition mathématique). Nous souhaitons maintenant générer des cartes non étiquetées.

Dans deux publications [Tutte, 1963, Tutte, 1968], W. Tutte a proposé une décomposition inductive des cartes planaires enracinées (non étiquetées). Par la suite, T. R. S. Walsh et A. B. Lehman ont décrit une décomposition des cartes enracinées indépendamment de leur genre, par effacement de leur arête portant le brin racine [Walsh et al., 1972a].

Nous dérivons de ces travaux une méthode de génération de cartes enracinées. Dans ce chapitre, nous formalisons les opérations de construction de cartes de genre quelconque de Walsh et Lehman. Dans le chapitre 9, nous explicitons un théorème de décomposition des cartes enracinées de genre quelconque, puis nous formalisons les opérations de construction de cartes planaires enracinées de Tutte et nous explicitons un théorème analogue pour les cartes planaires enracinées.

Nous utilisons une formalisation simplifiée des cartes étiquetées – les cartes locales, définies dans la partie 5.6 du chapitre 5 – encodables par une seule permutation.

Ce travail a été présenté durant la conférence Tests and Proofs (TAP) en juillet 2016 [Dubois et al., 2016].

Contrairement aux travaux présentés dans les autres chapitres de ce mémoire, cette étude utilise l'assistant de preuve Coq, assisté de méthodes de test. Nous avons vu dans le chapitre 5 que la vérification par preuve automatique de la transitivité d'un couple de permutations était problématique. Ceci nous a encouragé à faire usage d'un prouveur interactif. Cette nouvelle méthodologie, mêlant différentes techniques de test et des preuves interactives, est plus coûteuse en investissement mais permet d'élargir le cadre d'étude et de preuve formelle.

Dans cette étude, les opérations sur les permutations présentées dans le chapitre 4 jouent un rôle important. Nous formalisons en Coq les notions de permutation et de carte combinatoire, deux opérations sur les permutations, ainsi que deux opérations sur les cartes combinatoires. Nous définissons tout d'abord ces opérations sur les fonctions. Nous prouvons ensuite formellement que les deux premières opérations peuvent être restreintes aux permutations, et les deux dernières aux cartes. En d'autres termes, nous montrons qu'elles préservent respectivement les permutations et la structure des cartes.

A moins que la preuve d'un lemme ou théorème ne soit triviale, il est préférable de le tester avant de le prouver. Nous utilisons ici le test aléatoire (RT) et le test exhaustif borné (BET), décrits dans le chapitre 2. Un challenge pour le BET est de concevoir et d'implémenter des algorithmes efficaces générant des données. Nous l'abordons en bénéficiant des atouts de la programmation logique implémentée dans un système Prolog. La programmation logique est bien adaptée à la conception d'algorithmes de par sa proximité avec les spécifications en logique du premier ordre. Grâce au mécanisme de retour en arrière (*backtracking*), les prédicats caractéristiques écrits en Prolog peuvent souvent être utilisés comme générateurs exhaustifs, sans surcoût de développement.

Nous présentons une application du RT et du BET pour déboguer des spécifications Coq de structures combinatoires. Notre approche originale de deux études de cas (permutations et cartes) est également une contribution dans le domaine de la formalisation des mathématiques. En comparaison avec d'autres approches [Dubois et al., 2007, Gonthier, 2008, Dufourd, 2008], notre formalisation est très proche de la définition mathématique d'une carte, en tant que couple transitif de permutations.

Notre travail est disponible à l'adresse http://members.femto-st.fr/alain-giorgetti/en/coq-unit-testing. Il a été développé avec Coq 8.4 et SWI-Prolog 5.10.4. Le lecteur est supposé être familier avec la programmation logique et la programmation fonctionnelle. Dans le cas contraire, il peut lire un court résumé de connaissances basiques en Prolog dans [Giorgetti et al., 2012], ainsi qu'un bref tour d'horizon de la programmation en Coq dans [Bertot et al., 2004, Chapitre 2]. Les parties de code présentées sont accessibles car elles sont toutes commentées dans le texte.

La partie 8.1 présente la méthodologie de test sur un exemple simple portant sur les permutations. La partie 8.2 présente la formalisation Coq des cartes enracinées, des théorèmes de correction, ainsi que les tests effectués avant de les prouver. Des perspectives sont ensuite données dans la partie 8.3.

8.1 Test de conjectures Coq

Cette partie présente notre méthodologie pour tester des spécifications Coq. Avant d'investir du temps à essayer de prouver des lemmes potentiellement faux, nous souhaitons tester leur validité. Nous considérons ici deux manières de tester : le test aléatoire (dans la partie 8.1.2) et le test exhaustif borné (dans la partie 8.1.3). Ils sont illustrés par l'exemple fil-rouge des permutations sur un ensemble fini, présenté dans la partie 8.1.1.

8.1.1 Permutations en Coq

Les permutations sur un ensemble fini sont au cœur des définitions des cartes combinatoires. Nous avons pris le parti dans notre étude de les définir en tant qu'endofonctions injectives. À notre connaissance, aucune bibliothèque Coq ne définit les permutations de cette manière. Le listing 8.1 montre notre formalisation Coq des permutations.

Une permutation est définie comme une fonction injective d'un intervalle d'entiers naturels (dont la borne inférieure est 0) dans lui-même. En Coq, le type inductif `nat` des entiers naturels de Peano est prédéfini, avec les constructeurs `O` pour zero et `S` pour la fonction de succession. Nous manipulons les fonctions définies sur `nat` (appelées fonctions natu-

relles par la suite), mais nous imposons seulement des contraintes pour leurs valeurs sur un intervalle $\{0, \ldots, n-1\}$. Les prédicats `is_endo` et `is_inj` définissent respectivement les propriétés d'être une endofonction et d'injectivité sur cet intervalle. Ainsi une permutation est un enregistrement (Record) composé d'une fonction naturelle et de preuves que cette fonction satisfait les deux propriétés précédentes. Par commodité, nous considérons également la conjonction `is_permut` de ces deux propriétés.

```
Definition is_endo (n : nat) (f : nat → nat) :=
 ∀ x, x < n → f x < n.
Definition is_inj (n : nat) (f : nat → nat) := ∀ x y,
 x < n → y < n → x <> y → f x <> f y.
Record permut (n : nat) : Set := MkPermut {
 fct : nat → nat;
 endo : is_endo n fct;
 inj : is_inj n fct }.
Definition is_permut n f := is_endo n f ∧ is_inj n f.
```

Listing 8.1 – Permutations en tant qu'endofonctions injectives en Coq.

Nous pouvons définir un codage plus concret des permutations. Comme nous l'avons vu dans la partie 2.1.3.1, une permutation p sur $\{0, \ldots, n-1\}$ peut également être représentée par la liste $[p(0); p(1); \ldots; p(n-1)]$ de ses images. Par exemple, la liste $[1; 0; 3; 2; 6; 4; 5]$ représente la permutation $p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 3 & 2 & 6 & 4 & 5 \end{pmatrix}$. Nous allons générer les permutations en tant que listes. Nous passerons de cette représentation à la représentation fonctionnelle grâce à la fonction `list2fun` définie par

```
Definition list2fun (l:list nat) : nat → nat :=
 fun (n:nat) ⇒ nth n l n.
```

La fonction `nth` de la bibliothèque standard de Coq est telle que `(nth n l d)` retourne le `n`-ième élément de la liste `l` s'il existe, et `d` sinon.

Nous implémentons en Coq une extension à tous les naturels des fonctions *insert* et *sum* définies dans le chapitre 4.

```
Definition insert_fun n (f : nat → nat) (i : nat) : nat → nat :=
 fun x ⇒ if le_lt_dec i n then
   match nat_compare x n with
     Eq ⇒ i
   | Lt ⇒ if eq_nat_dec (f x) i then n else f x
   | Gt ⇒ x
   end
  else x.
```

Listing 8.2 – Opération d'insertion dans une fonction naturelle.

D'après la définition 15, l'insertion avant un entier naturel i dans une fonction définie sur $\{0, \ldots, n-1\}$ est la fonction f' définie sur $\{0, \ldots, n\}$ par :

(a) $f' = f$ si $i > n$;
(b) $f' = f$ sur $\{0, \ldots, n-1\}$, étendue avec le point fixe $f'(n) = n$ si $i = n$;
(c) $f'(n) = i$ si $i < n$, $f'(j) = n$ si $f(j) = i$, et $f'(j) = f(j)$ si $0 \leq j \leq n-1$ et $f(j) \neq i$.

Le listing 8.2 présente une extension de l'opération d'insertion à toute fonction naturelle, en Coq. Nous utilisons ici les opérateurs arithmétiques `le_lt_dec`, `nat_compare` et

eq_nat_dec de comparaison et d'égalité de la bibliothèque standard de Coq. Les expressions (le_lt_dec i n) et (eq_nat_dec y i) formalisent respectivement $i \leq n$ et $y = i$. L'expression (nat_compare x n) vaut Eq si $x = n$, Lt si $x < n$ et Gt si $x > n$. Ce troisième cas étend l'opération d'insertion à tous les entiers naturels.

D'après la définition 17, la somme directe d'une fonction f_1 définie sur $\{0, \ldots, n_1 - 1\}$ et d'une fonction f_2 définie sur $\{0, \ldots, n_2 - 1\}$ est la fonction f sur $\{0, \ldots, n_1 + n_2 - 1\}$ telle que $f(x) = f_1(x)$ si $0 \leq x < n_1$ et $f(x) = f_2(x - n_1) + n_1$ si $n_1 \leq x < n_1 + n_2$. Une extension de cette somme aux fonctions naturelles est définie en Coq par

```
Definition sum_fun n1 f1 n2 f2 : nat → nat :=
 fun x ⇒
  if lt_ge_dec x n1 then
   f1 x
 else
  if lt_ge_dec x (n1+n2) then
   (f2 (x-n1)) + n1
  else x.
```

L'expression (le_ge_dec x y) formalise $x \geq y$.

Les lemmes du listing 8.3 énoncent que ces deux opérations préservent les permutations. Pour valider ces lemmes, nous définissons des versions booléennes is_endob, is_injb et is_permutb des propriétés logiques is_endo, is_inj et is_permut. Le listing 8.4 montre les fonctions is_endob, is_permutb is_injb.

```
Lemma insert_permut: ∀ (n : nat) (p : permut n) (i : nat),
 is_permut (S n) (insert_fun n (fct p) i).
Lemma sum_permut: ∀ n1 (p1 : permut n1) n2 (p2 : permut n2),
 is_permut (n1 + n2) (sum_fun n1 (fct p1) n2 (fct p2)).
```

Listing 8.3 – Propriétés de préservation des opérations d'insertion et de somme.

```
Fixpoint is_endob_aux n f m := match m with
  0 ⇒ si (lt_dec (f 0) n) then true else false
| S p ⇒ si (lt_dec (f m) n) then is_endob_aux n f p else false
end.
Definition is_endob n f := match n with
  0 ⇒ true
| S p ⇒ is_endob_aux n f p
end.
Lemma is_endo_dec : ∀ n f, (is_endob n f  = true ↔ is_endo n f).
Definition is_permutb n f := (is_endob n f) && (is_injb n f).
Definition is_injb n f := @uniq nat eq_nat_dec (map f (seq 0 n)).
```

Listing 8.4 – Fonctions booléennes pour les permutations.

Une évaluation de (is_endob n f) retourne true si et seulement si la fonction f est une endofonction sur $\{0, \ldots, n-1\}$. Le lemme is_endo_dec établit que la fonction booléenne is_endob est une implémentation correcte du prédicat is_endo. Des lemmes similaires sont prouvés pour les deux autres fonctions booléennes. Si la corrélation entre is_endo et is_endob est quasi-immédiate, ce n'est pas le cas pour is_inj et is_injb. Pour définir is_injb, nous utilisons un autre lemme uniq que nous avons prouvé : une fonc-

tion f est injective sur $\{0, 1, \ldots, n\}$ si et seulement si la liste $[f(0); f(1); \ldots; f(n)]$ de ses images n'a pas de doublon.

Dans ce travail, nous avons choisi de formaliser les permutations par des fonctions Coq plutôt que par des listes Coq afin de traduire fidèlement en Coq les définitions mathématiques des opérations sur les permutations , en écrivant `(f x)` pour une fonction Coq `f` plutôt que `(nth l x d)` pour une liste Coq `l`. Cette dernière expression pose le problème du choix d'une valeur par défaut `d` quand `x` atteint ou dépasse la longueur de la liste `l`. De plus, son utilisation fournirait des implémentations inefficaces des opérations sur les permutations. Des efforts de conception et de vérification supplémentaires sont nécessaires pour définir inductivement l'insertion et la somme directe sur des listes sans utiliser `nth`.

8.1.2 Test aléatoire

QuickChick [Hritcu et al., 2015] est un greffon de test aléatoire pour Coq. Il permet de vérifier la validité de conjectures exécutables en générant des données aléatoires. QuickChick est principalement une plateforme générique fournissant des outils combinatoires pour écrire du code de test, en particulier des générateurs aléatoires. Voici la procédure générale que nous suivons pour valider par test une conjecture comme

```
∀ x: T, precondition x → conclusion (f x),
```

où `precondition` et `conclusion` sont des prédicats logiques. Nous définissons d'abord un générateur aléatoire `gen_T` de valeurs de type `T` qui satisfont la propriété `precondition`. Nous transformons ensuite la `conclusion` en une fonction booléenne `conclusionb` – si c'est possible, sinon QuickChick ne peut pas fonctionner – que nous prouvons sémantiquement équivalente au prédicat logique. Le test est exécuté en utilisant la commande

```
QuickCheck (forAll gen_T (fun x ⇒conclusionb (f x)).
```

qui génère un nombre fixe d'entrées en utilisant le générateur `gen_T`, et qui applique pour chacune d'entre elles la fonction `f` et vérifie la propriété à tester (`conclusion`).

Dans notre approche, nous supposons que le générateur est correct. QuickChick propose des théorèmes (ou axiomes) qui pourraient être utilisés pour prouver que le générateur est correct, mais c'est une tâche difficile. Dans la suite, nous proposons de tester que le générateur produit des sorties correctes. À cette fin, nous mettons en œuvre la même approche : transformer la propriété logique `precondition` en une propriété exécutable `preconditionb`.

Nous illustrons maintenant les possibilités de QuickChick sur les permutations codées sous forme de fonctions naturelles. QuickChick ne permet pas de générer des fonctions ; aussi nous générons les permutations en tant que listes, puis nous les transformons en fonctions, comme détaillé dans la partie 8.1.1.

Nous définissons tout d'abord un générateur de permutations sur $\{0, \ldots, n-1\}$, en tant que listes sans doublon contenant $0, 1, \ldots n-1$ dans tous les ordres possibles. Ce générateur est présenté dans le listing 8.5.

```
Fixpoint gen_permutl (n : nat) : G (list nat) := match n with
  0   ⇒ returnGen nil
| S p ⇒ do! m ← choose (0, n);
```

```
   liftGen (insert_pos p m) (gen_permutl p)
end.
```

Listing 8.5 – Générateur de permutations.

Si `n = 0`, la sortie est la liste vide. Sinon, `n` est le successeur d'un entier naturel `p`. L'appel récursif (`gen_permutl p`) génère une liste codant une permutation sur $\{0, \ldots,$ `p`$-1\}$. Le combinateur `liftGen` applique à cette liste la fonction `insert_pos p m`, pour y insérer la valeur `p` à une position `m` qui est choisie aléatoirement (en utilisant le combinateur `choose`).

Pour avoir confiance dans ce générateur, nous testons que les sorties ne contiennent aucun doublon, que leur longueur est `n` et que tous leurs éléments sont des entiers naturels inférieurs à `n` (ces trois conditions sont implémentées par un prédicat booléen `list_permutb`).

La commande suivante génère 10000 tests, et le résultat de ces tests est donné.

```
QuickCheck (sized (fun n ⇒
 forAll (gen_permutl n) (list_permutb n))).
+++ OK, passed 10000 tests
```

Le nombre maximal de tests peut être ajusté par l'utilisateur. Nous itérons le processus pour différentes valeurs de `n` grâce au combinateur `sized`. Le combinateur `sized` prend en argument un générateur paramétré par une taille (ici `n`) et fournit un générateur non paramétré, en choisissant des tailles.

Nous pouvons suivre le même processus pour valider que les permutations en tant que fonctions naturelles sont obtenues par application de la fonction de traduction `list2fun` sur les listes générées par le générateur précédent `gen_permutl` :

```
Definition fun_permutb n l := is_permutb n (list2fun l).
QuickCheck (sized (fun n ⇒
 forAll (gen_permutl n) (fun_permutb n))).
+++ OK, passed 10000 tests
```

Nous sommes maintenant prêts à tester les conjectures formulées dans le listing 8.3, en suivant la même méthodologie : (i) Pour générer une fonction naturelle représentant une permutation, nous utilisons le générateur de listes `gen_permutl` ; (ii) la propriété logique à tester est remplacée par une version booléenne. Cette dernière propriété est obtenue par la fonction de traduction `list2fun` Par exemple, un test du lemme `insert_permut` s'obtient par la commande

```
QuickCheck (sized (fun n ⇒ forAll (gen_permutl n)
              (fun l ⇒ (forAll arbitraryNat
                (fun i ⇒ let f := list2fun l in
                  is_permutb (S n) (insert_fun n f i)))))).
```

Cette commande `QuickCheck` a la même structure que les précédentes, sauf que nous utilisons deux générateurs, un pour les permutations et un autre pour les entiers naturels arbitraires, nommé `arbitraryNat`. Cette commande génère 10000 tests. Si nous injectons une erreur dans la définition de `insert_fun` (présentée dans le listing 8.2), en remplaçant le résultat `n` par `S n` dans le cas `Lt`, nous obtenons un contre-exemple `l` = $[0; 1]$ et `i` = 0 pour `n` = 2. En effet, dans ce cas `insert_fun` insère la valeur `3` dans `f`, ce qui donne la liste = $[3; 1; 0]$ qui ne représente pas une permutation.

8.1.3 Test exhaustif borné

Pour tester les spécifications Coq, nous utilisons également le test exhaustif borné (BET) et son support en programmation logique, pour de nombreuses raisons. Tout d'abord, le BET est particulièrement bien adapté à la combinatoire énumérative, parce qu'il correspond à l'activité de génération d'objets combinatoires, classique dans ce domaine. Deuxièmement, le BET fournit à l'auteur d'un lemme erroné la plus petite structure combinatoire révélant son erreur. Troisièmement, les structures combinatoires formalisées en Coq en tant que structures inductives avec propriétés sont souvent faciles à formaliser en logique du premier ordre avec des prédicats Prolog. Quatrièmement, le mécanisme de *backtracking* (retour en arrière) de Prolog fournit souvent des générateurs exhaustifs bornés sans coût supplémentaire. Tous ces avantages sont illustrés dans ce chapitre. Nous présentons d'abord les principes du BET utilisant la programmation logique.

Afin de rendre les tâches de validation plus faciles, nous utilisons une bibliothèque de validation Prolog créée par Valerio Senni [Senni, 2012] et appliquée en 2012 à la validation d'algorithmes sur les mots (présentés dans le chapitre 6) codant les cartes planaires enracinées [Giorgetti et al., 2012]. La bibliothèque fournit une automatisation complète pour la comparaison exhaustive bornée symétrique, c.a.d. la comparaison des objets générés par deux générations exhaustives bornées différentes. Elle renvoie des contre-exemples si la validation échoue (de sorte que le processus de débogage est guidé par ces contre-exemples), et elle recueille des statistiques telles que le temps de génération et la consommation de mémoire. Nous illustrons quelques-unes des caractéristiques de la bibliothèque de validation sur l'exemple des permutations.

Nous encodons une fonction f sur $\{0,\dots,n-1\}$ par la liste Prolog `[`$f(0)$`,...,`$f(n-1)$`]` de ses images (notation sur une ligne). Une liste est linéaire si elle n'a pas de doublon. Le listing 8.6 montre un prédicat Prolog `line` tel que la formule `line(L,N)` (resp. `line(L,K,N)`) est vraie si et seulement si `L` est une liste linéaire de longueur `N` (resp. `K`) dont les éléments sont dans {`0`,...,`N-1`}. Nous disons alors que `L` est une liste de permutation. En d'autres termes, ce prédicat caractéristique des permutations correspond au prédicat Coq `is_permut`.

La formule `in(K,I,J)` est vraie si et seulement si l'entier `K` est dans l'intervalle [`I`..`J`]. La définition du prédicat `line` est basée sur le principe suivant. Si `P` est une liste linéaire de longueur `K-1` avec ses éléments dans {`0`,...,`N-1`}, alors la liste obtenue en plaçant une valeur `Y` comprise entre `0` et `N-1` (qui n'est pas déjà dans `P`) en tête de `P` est également linéaire.

```
line([],0,_).
line([Y|P],K,N) :- K > 0, Km1 is K-1, Nm1 is N-1, in(Y,0,Nm1),
 line(P,Km1,N), \+ member(Y,P).
line(P,N) :- line(P,N,N).
```

Listing 8.6 – Permutations en Prolog.

Un avantage certain de la programmation logique est que le prédicat `line` a deux fonctions : *accepteur* de listes de permutation, et *générateur* énumérant les listes de permutation d'une taille donnée. Pour un prédicat caractéristique p et une taille donnée n, le schéma de requête

$$:-\ p(\mathtt{L},n),\ \mathtt{write_coq(L)},\ \mathtt{fail}. \qquad (Q)$$

permet l'énumération de toutes les données acceptées de taille n. En effet, la requête (Q)

force la construction d'une première donnée `L` de taille n acceptée par p, son traitement (par `write_coq`), et l'échec du mécanisme de preuve en utilisant la fonction native `fail`. Si la preuve échoue, le mécanisme de *backtracking* retrouve le dernier "point de choix" (nécessairement dans p) et déclenche la génération d'une nouvelle donnée jusqu'à ce qu'il n'y ait plus de point de choix. Ici, le prédicat `write_coq` est défini par l'utilisateur pour générer en sortie (comme effet de bord) un cas de test dans la syntaxe Coq. Par exemple, il peut facilement être défini de telle sorte que la requête

```
:- line(L,3), write_coq(3), fail.
```

écrive une ligne de Coq telle que

```
Eval compute in (is_permutb 3 (list2fun [2;0;1])).
```

pour chaque liste de permutation de longueur 3. Chaque ligne constitue un cas de test pour la fonction Coq `is_permutb`, en supposant que la fonction Coq `list2fun` et le programme Prolog présenté dans le listing 8.6 sont corrects. Ce dernier peut être vérifié de deux manières : par l'inspection visuelle des listes qu'il génère, ou par comptage. Pour le comptage, la bibliothèque fournit le prédicat `iterate` tel que la requête

```
:- iterate(0,6,line).
```

produit en sortie les nombres 1, 1, 2, 6, 24, 120 et 720 de listes distinctes de longueur de 0 à 6 acceptées par le prédicat `line`. Nous reconnaissons les premiers nombres $n!$ de permutations de longueur n.

Nous pouvons maintenant adapter le prédicat `write_coq` de la requête (Q) pour le BET des lemmes présentés dans le listing 8.3. Pour le lemme `insert_permut`, l'évaluation de la requête peut générer dans un fichier Coq toutes les lignes de la forme

`Eval compute in (is_permutb` $(n+1)$ `(insert_fun` n `(list2fun` l`)` i`)).`

pour n jusqu'à une limite donnée, pour $0 \leq i \leq n$ et pour toute liste l (de longueur n) satisfaisant `line(`l`,`n`)`. Ensuite, nous vérifions que la compilation du fichier Coq généré produit toujours `true`. Nous procédons de même avec le lemme `sum_permut`.

Comme mentionné précédemment à propos de la propriété `is_inj`, il peut être difficile d'écrire une version booléenne d'une propriété et de prouver sa correction. Dans ce cas le BET reste parfois possible, comme illustré par l'exemple suivant. Supposons que nous ne trouvions pas l'implémentation de la propriété `is_permut`. Nous générons alors une preuve non-calculatoire généralisant l'exemple suivant

```
Goal is_permut 3 (list2fun (2::0::1::nil)). unfold is_permut.
unfold list2fun. unfold list2funX. split.
- unfold is_endo. intros x Hx. assert (x = 0 ∨ x = 1 ∨ x = 2).
   omega.
  firstorder; subst; simpl; omega.
- unfold is_inj. intros x y Hx Hy Hxy.
  assert (x = 0 ∨ x = 1 ∨ x = 2). omega.
  assert (y = 0 ∨ y = 1 ∨ y = 2). omega.
  firstorder; subst; simpl; omega. Qed.
```

La preuve se subdivise tout d'abord en une sous-preuve de la propriété `is_endo` et une sous-preuve de la propriété `is_inj`. Chaque sous-preuve s'obtient par énumération de toutes les valeurs possibles de `x` (et `y` pour l'injectivité). Cette approche convient chaque fois que la propriété est universellement quantifiée une variable `i` de type `nat` bornée

par un nombre b. La tactique

```
assert (i = 0 ∨ i = 1...∨ i = b)
```

énumère alors toutes les valeurs possibles de i. L'assertion est prouvée par la tactique omega qui implémente une procédure de décision pour l'arithmétique linéaire (le test Omega [Pugh, 1991]). La preuve est ensuite décomposée en cas par la tactique firstorder. Dans la sous-preuve pour l'injectivité, chaque cas contient des hypothèses x = ... et y = ... affectant des valeurs aux deux variables. Après remplacement de x et y par leurs valeurs, le test Omega termine la preuve.

Afin de mesurer le temps requis pour répondre à une requête (par exemple le temps pour générer l'ensemble des mots d'une taille donnée), il est possible d'utiliser le schéma de requête suivant :

```
:- statistics(runtime,[T1,_]),  Goal, statistics(runtime,[T2,_]),
   Time is T2-T1.
```

où la fonction native statistics est utilisée pour mesurer le temps CPU avant et après l'exécution du *Goal*.

8.2 ETUDE DE CAS DES CARTES LOCALES

Nous appliquons maintenant notre méthodologie de test à des théorèmes Coq plus ambitieux. En tant qu'étude de cas, nous considérons la famille combinatoire des cartes locales, formalisées en Coq en tant que permutations transitives (partie 8.2.1). Nous introduisons ensuite deux opérations permettant de construire une carte à partir d'une ou deux cartes plus petites par addition d'une arête (partie 8.2.2). Ces deux opérations sont définies à partir des deux opérations d'insertion et de somme directe définies et implémentées en C dans le chapitre 4, et définies en Coq dans la partie 8.1. Enfin, nous validons par test puis prouvons formellement que les deux opérations préservent les permutations et la transitivité (partie 8.2.3). La partie 8.2.4 fournit des statistiques sur les tests et les preuves effectués.

8.2.1 DÉFINITIONS ET FORMALISATION

Nous implémentons d'abord en Coq l'involution sans point fixe locale (définition 18 du chapitre 5) sur les entiers naturels, par la définition opp.

```
Definition opp (n:nat) : nat :=
 if (even_odd_dec n) then S n else (n-1).
```

Nous définissons ensuite la transitivité d'une fonction f sur un ensemble D de telle sorte qu'un triplet (D, R, opp) soit une carte locale si et seulement si sa rotation R est transitive. Nous disons qu'il y a un pas entre deux éléments x et y de D par une fonction f si et seulement si $f(x) = y$, $f(y) = x$ ou $\mathrm{opp}(x) = y$. Deux nombres x et y sont connectés par f si et seulement s'il existe un chemin (i.e. une séquence de pas) de x à y. Enfin, une fonction f est transitive sur D si deux éléments quelconques de D sont connectés par f.

```
Inductive connected n (f : nat → nat) : nat → nat → nat → Prop :=
 | c0    : ∀ x y, x < n → y < n → x = y → connected n f 0 x y
```

```
 | cfirst : ∀ l x y z, x < n → y < n → z < n →
    f x = y ∨ f y = x ∨ opp x = y →
    connected n f l y z → connected n f (S l) x z.
Definition is_transitive_fun (n: nat) (f : nat → nat) : Prop :=
 ∀ x, x < n → ∀ y, y < n → ∃ m, connected n f m x y.
Definition transitive_fun (n: nat) (f : nat → nat) : Prop :=
 ∀ y, y < n → ∀ x, x < y → ∃ m, connected n f m x y.
Definition is_transitive n (p : permut n) :=
 transitive_fun n (fct p).
```

Listing 8.7 – Définition de la transitivité.

Le listing 8.7 montre une formalisation Coq de la transitivité d'une fonction naturelle et d'une permutation sur $\{0,\ldots,n-1\}$. La propriété de connectivité est spécifiée par le prédicat inductif `connected` de telle manière que `(connected n f l x y)` est vraie si et seulement si les entiers naturels `x` et `y` sont reliés par exactement `l` pas de `f`. Le constructeur `cfirst` établit qu'un pas entre `x` et `y` peut être décomposé en son premier pas et sa fin, tandis que le constructeur `c0` exprime le cas trivial où $x = y$. Cette définition est complétée par trois lemmes présentés dans le listing 8.8 : un lemme `clast` décomposant un chemin en son début et son dernier pas, et deux lemmes (`connected_sym` et `connected_trans`) prouvant respectivement la symétrie et la transitivité de la relation binaire `(connected n f l)`.

```
Lemma clast (n : nat) (f : nat → nat) : ∀ l,
 ∀ x y z, x < n → y < n → z < n →
 connected n f l x y → (f y = z ∨ f z = y ∨ opp y = z) →
 connected n f (S l) x z.
Lemma connected_sym n f l : ∀ x, x < n → ∀ y, y < n →
 connected n f l x y → connected n f l y x.
Lemma connected_trans n f l1 l2 :
 ∀ x, x < n → ∀ y, y < n → ∀ z, z < n →
 connected n f l1 x y → connected n f l2 y z →
 connected n f (l1+l2) x z.
```

Listing 8.8 – Lemmes sur la propriété de connectivité.

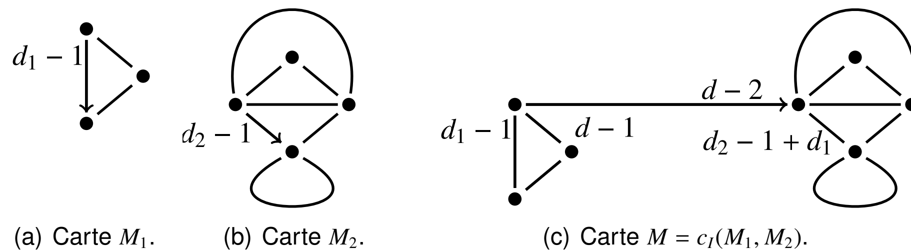
Les définitions `is_transitive_fun` et `transitive_fun` implémentent deux versions de la propriété de transitivité d'une fonction. Les décompositions en cas de certaines preuves sont considérablement raccourcies par l'utilisation de la seconde définition `transitive_fun` qui ne considère que les nombres `x` strictement inférieurs à `y`. En utilisant la symétrie du prédicat `connected`, nous avons démontré que les deux définitions de transitivité sont équivalentes. Le prédicat `is_transitive` définit la transitivité d'une permutation comme la transitivité de sa fonction associée.

Toutes les cartes considérées ci-après sont locales et sont codées par leur rotation transitive. Une carte locale (D, R, opp) possédant e arêtes est formalisée en Coq par une structure Record composée de sa permutation des sommets de longueur $2e$ (sa rotation) et d'une preuve que cette permutation est transitive, de la manière suivante :

```
Record carte (e : nat) : Set := {
 rotation : permut (2*e);
 transitive : is_transitive rotation }.
```

8.2.2 Opérations de construction de cartes

Une arête est un isthme si ses deux brins associés sont incidents à la même face. Une carte (différente de la carte sommet) est isthmique (resp. non isthmique) si son arête racine est (resp. n'est pas) un isthme. Nous définissons ici une opération c_I construisant une carte isthmique à partir de deux cartes, et une famille d'opérations c_N^k (indexées par un entier k) construisant une carte non isthmique à partir d'une seule carte. Ces deux opérations procèdent par addition d'une arête. Nous expliquons leurs définitions uniquement pour des nombres pairs de brins, bien qu'elles soient définies pour des nombres quelconques de brins.



(a) Carte M_1. (b) Carte M_2. (c) Carte $M = c_I(M_1, M_2)$.

Figure 8.1 – Exemple de construction d'une carte isthmique.

8.2.2.1 Opération isthmique

L'opération c_I est illustrée par un exemple dans la figure 8.1. Elle ajoute une arête isthmique entre une carte locale M_1 avec e_1 arêtes et une carte locale M_2 avec e_2 arêtes. Le résultat est une carte $M = c_I(M_1, M_2)$ avec $e_1 + e_2 + 1$ arêtes. Soit $d_1 = 2e_1$, $d_2 = 2e_2$ et $d = 2e_1 + 2e_2 + 2$ les nombres de brins de M_1, M_2 et M. L'arête additionnelle est composée des deux brins $d - 1$ $(= d_1 + d_2 + 1)$ et $d - 2$ $(= d_1 + d_2)$. La racine de M est le brin $d - 1$, et son brin opposé opp$(d - 1)$ est $d - 2$.

Le listing 8.9 présente une fonction Coq `isthmic_fun` qui implémente cette opération sur deux fonctions naturelles `r1` et `r2` représentant les rotations R_1 et R_2 de M_1 et M_2. Les deux brins sont insérés grâce à deux applications successives de la fonction `insert_fun`. Si R_1 et R_2 représentent la carte vide ($d_1 = d_2 = 0$), la carte résultante M a une arête (qui n'est pas une boucle) et la rotation résultante R est la permutation $(0)\,(1)$. Si M_1 est la carte vide ($d_1 = 0$) et M_2 n'est pas vide ($d_2 \neq 0$), le brin $d - 2$ $(= d_2)$ est inséré juste avant le brin $d_2 - 1$ dans son cycle de R_2, puis le brin $d - 1$ $(= d_2 + 1)$ est ajouté en tant que point fixe de R. Si M_1 n'est pas vide ($d_1 \neq 0$) et M_2 est la carte vide ($d_2 = 0$), le brin $d - 2$ $(= d_1)$ est ajouté en tant que point fixe, puis le brin $d - 1$ $(= d_1 + 1)$ est inséré juste avant le brin $d_1 - 1$ dans son cycle. Sinon, si ni M_1 ni M_2 n'est la carte vide, la somme directe $R' = sum(R_1, R_2)$ est construite grâce à une application de la fonction `sum_fun`. Le brin $d_1 + d_2$ est alors inséré juste avant le brin $d_2 - 1 + d_1$ dans son cycle de R', et le brin $d_1 + d_2 + 1$ est inséré juste avant le brin $d_1 - 1$ dans son cycle de la permutation résultante.

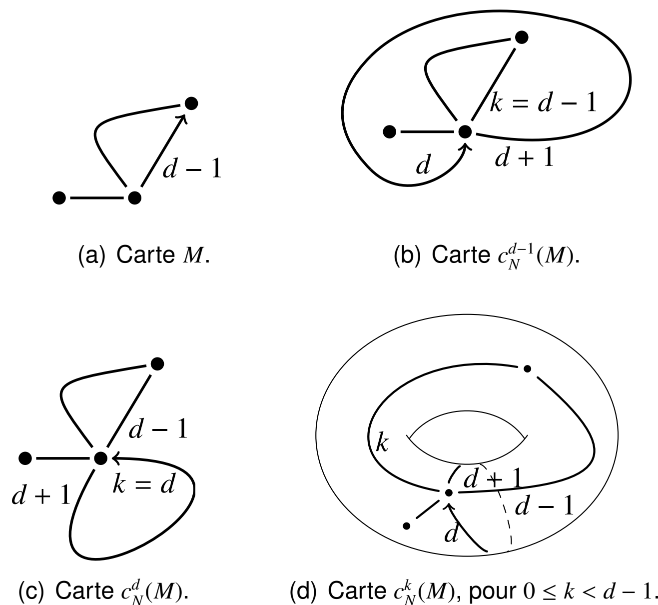
```
Definition isthmic_fun d1 (r1 : nat → nat)
                          d2 (r2 : nat → nat) : nat → nat :=
match d1 with
| 0 ⇒
```

```
    match d2 with
    | 0 ⇒ insert_fun 1 (insert_fun 0 r2 0) 1
    | S d2m1 ⇒ insert_fun (d2+1) (insert_fun d2 r2 d2m1) (d2+1)
    end
| S d1m1 ⇒
    match d2 with
    | 0 ⇒ insert_fun (d1+1) (insert_fun d1 r1 d1) d1m1
    | S d2m1 ⇒
       insert_fun (d1+d2+1)
        (insert_fun (d1+d2) (sum_fun d1 r1 d2 r2) (d1m1+d2))
       d1m1
    end
end.
```

Listing 8.9 – Opération isthmique en Coq.

8.2.2.2 Opération non isthmique

Pour $0 \leq k \leq 2e$, l'opération c_N^k ajoute une arête non isthmique dans une carte locale M possédant e arêtes (représentée par sa rotation R de longueur $d = 2e$) pour obtenir une carte locale M' avec $e + 1$ arêtes représentée par sa rotation R' de longueur $d' = 2e + 2$. La permutation résultante R' est obtenue par insertion de la nouvelle racine $d + 1$ et de



(a) Carte M. (b) Carte $c_N^{d-1}(M)$.

(c) Carte $c_N^d(M)$. (d) Carte $c_N^k(M)$, pour $0 \leq k < d - 1$.

Figure 8.2 – Exemples de construction de cartes non isthmiques.

son opposé d dans R. Si M est la carte vide, la nouvelle arête est ajoutée – en tant que boucle – de manière unique pour obtenir M'. Sinon, il existe $d + 1$ manières d'ajouter cette nouvelle arête, distinguées par les valeurs de k entre 0 et d. La figure 8.2 montre une carte M avec trois arêtes (dans la figure 8.2(a)) et trois cartes obtenues par application de l'opération c_N^k sur M, pour différentes valeurs de k. Les deux premières sont des cartes planaires, tandis que la dernière (dans la figure 8.2(d)) est une carte sur un tore.

Si $k = d - 1$ et $k = d$, l'arête ajoutée est une boucle. Ces cas sont respectivement illustrés dans les figures 8.2(b) et 8.2(c). Notons que l'ordre d'insertion des brins est important : dans la figure 8.2(b), le brin d est inséré juste avant le brin $d-1$ puis le brin $d+1$ est inséré juste avant le brin $k = d - 1$, de sorte que le brin d se retrouve finalement juste avant le brin $d + 1$ dans son cycle de la rotation R' de M'. Dans tous les autres cas, le brin d est juste avant le brin $d - 1$ dans R'. La figure 8.2(d) montre un cas où le brin k $(0 \leq k < d - 1)$ n'est pas incident à la même face que le brin $d - 1$. Dans ce cas, la nouvelle arête ne peut être ajoutée que grâce à un trou perforant la surface.

Le listing 8.10 présente une fonction Coq `non_isthmic_fun` implémentant cette opération sur une fonction naturelle `r` représentant la permutation des sommets R d'une carte locale ayant d brins, avec $k \leq d$. Si $d = 0$, la rotation R représente la carte vide, la nouvelle arête est une boucle et la fonction résultante est la permutation $(0\ 1)$. Sinon, le brin d est inséré juste avant la racine $d - 1$ de M dans son cycle de R. Notons Q la permutation résultante. Le brin $d + 1$ est alors inséré juste avant le brin k dans son cycle de Q.

```
Definition non_isthmic_fun (d:nat) (r:nat → nat) (k:nat)
 (k_le_d:k ≤ d) : nat → nat :=
 match d with
 | 0     ⇒ insert_fun 1 (insert_fun 0 r 0) 0
 | S dm1 ⇒ insert_fun (S d) (insert_fun d r dm1) k
 end.
```

Listing 8.10 – Opération non isthmique en Coq.

8.2.3 Tests et preuves

Nous avons formalisé séparément les cartes locales et deux générations de construction de cartes, en tant qu'opérations sur les fonctions naturelles. Il reste à prouver leur correction, c.a.d. que chaque opération préserve les permutations transitives. Nous procédons en deux étapes. La première étape, présentée dans la partie 8.2.3.1, consiste à tester, puis à prouver que les deux opérations préservent les permutations. La deuxième étape, détaillée dans la partie 8.2.3.2, montre qu'elles préservent la transitivité.

8.2.3.1 Préservation des permutations

La preuve que les opérations de construction préservent les permutations est décomposée en lemmes intermédiaires. Les lemmes

```
Lemma isthmic_endo : ∀ d1 (r1:permut d1) d2 (r2:permut d2),
 is_endo (S (S (d1 + d2))) (isthmic_fun d1 (fct r1) d2 (fct r2)).
```

et

```
Lemma isthmic_inj : ∀ (d1 : nat) (r1 : permut d1)
                      (d2 : nat) (r2 : permut d2),
 is_inj (S (S (d1+d2))) (isthmic_fun d1 (fct r1) d2 (fct r2)).
```

établissent respectivement que l'opération isthmique préserve les endofonctions et l'injectivité. De même, les lemmes

```
Lemma non_isthmic_endo : ∀ (d : nat) (r : permut d)
                                (k : nat) (k_le_d : k ≤ d),
 is_endo (S (S d)) (@non_isthmic_fun d (fct r) k k_le_d).
```

et

```
Lemma non_isthmic_inj: ∀ d (r:permut d) k (k_le_d:k ≤ d),
 is_inj (S (S d)) (non_isthmic_fun d (fct r) k k_le_d).
```

établissent respectivement que l'opération non isthmique préserve les endofonctions et l'injectivité.

Comme dans la partie 8.1.3, nous validons par BET que les opérations isthmique et non isthmique préservent les permutations. Dans le cas non isthmique, nous rencontrons une spécificité du test avec des types dépendants. L'opération non isthmique est en effet paramétrée par une preuve. Ainsi, le BET doit générer une telle preuve pour chaque donnée de test. Dans le cas présent, nous avons besoin d'une preuve de $x \leq y$ pour chaque couple d'entiers naturels x et y. Le prédicat Coq ≤ est sémantiquement équivalent à la fonction booléenne `leb:nat→nat→bool`. La librairie standard de Coq contient le lemme

```
Lemma leb_complete : ∀ m n, leb m n = true → m ≤ n.
```

Ainsi, le terme (`leb_complete` x y `eq_refl`) est une preuve Coq de $x \leq y$. Dans ce terme, `eq_refl` est une preuve qu'un nombre est égal à lui-même. Pour la préservation des permutations par l'opération non isthmique, le BET génère des cas de test tels que

```
Eval compute in (
 let proof := leb_complete 2 3 eq_refl in
 is_permutb 5 (non_isthmic_fun 3 (list2fun [0;0;0]) 2 proof)).
```

pour $x = 2$ et $y = 3$. Après cette validation, nous avons prouvé tous les lemmes de préservation des permutations, principalement en dépliant les définitions.

```
Definition non_isthmic_permut (d : nat) (r : permut d)
 (k : nat) (k_le_d : k ≤ d) : permut (S (S d)) :=
 @MkPermut
  (S (S d))
  (@non_isthmic_fun d (fct r) k k_le_d)
  (non_isthmic_endo r k_le_d)
  (non_isthmic_inj r k_le_d).
```

Listing 8.11 – Définition d'une permutation non isthmique.

Avec ces lemmes, nous pouvons restreindre les opérations de construction aux permutations et donc définir les notions de permutation isthmique et non isthmique. Le listing 8.11 montre la définition de l'opération de construction `non_isthmic_permut` pour les permutations non isthmiques, basée sur le constructeur `MkPermut` des permutations défini dans le listing 8.1.1. Cette opération est définie à partir de l'opération `non_isthmic_fun` sur les fonctions naturelles et des preuves qu'elle préserve les endofonctions et l'injectivité (établies grâce aux lemmes `non_isthmic_endo` et `non_isthmic_inj`). Le cas isthmique est similaire.

Le test aléatoire nous permet également de valider la préservation des permutations par les opérations isthmique et non isthmique. La commande `QuickCheck` suivante génère

aléatoirement un premier entier naturel `d1`, une liste de permutation `l1` de longueur `d1` puis un second entier `d2` ainsi qu'une liste de permutation de longueur `d2`, puis vérifie si l'opération isthmique construit une permutation à partir des fonctions naturelles correspondantes.

```
QuickCheck (forAll arbitraryNat (fun d1 ⇒
 forAll (gen_permutl d1) (fun l1 ⇒ let f1 := list2fun l1 in
  forAll arbitraryNat (fun d2 ⇒
    forAll (gen_permutl d2) (fun l2 ⇒ let f2 := list2fun l2 in
     is_permutb (S (S (d1 + d2))) (isthmic_fun d1 f1 d2 f2)))))).
```

Pour l'opération non isthmique, le test aléatoire est similaire.

8.2.3.2 Préservation de la transitivité

Nous testons puis démontrons que les opérations isthmique et non isthmique préservent les permutations transitives et peuvent donc être considérées comme des opérations sur les cartes locales. Ces propriétés sont formalisées par les deux théorèmes présentés dans le listing 8.12. Le théorème `isthmic_trans` (resp. `non_isthmic_trans`) établit que l'opération isthmique (resp. non isthmique) préserve la transitivité lorsqu'elle agit sur deux permutations (resp. une permutation) de longueur paire.

```
Theorem isthmic_trans: ∀ d1 (r1: permut d1) d2 (r2: permut d2),
 even d1 → even d2 → is_transitive r1 → is_transitive r2 →
  is_transitive (isthmic_permut r1 r2).
Theorem non_isthmic_trans: ∀ d (r: permut d) k (k_le_d: k ≤ d),
 even d → is_transitive r → is_transitive (non_isthmic_permut r
    k_le_d).
```

Listing 8.12 – Préservation de la transitivité par les opérations isthmique et non isthmique.

Notons que ces théorèmes peuvent être étendus aux fonctions naturelles car les faits que `r1`, `r2` et `r` soient des endofonctions et injectives ne sont pas utilisés pour les prouver.

Pour le test, nous proposons dans le listing 8.13 une implémentation du prédicat de transitivité défini dans le listing 8.7. Il est basé sur une recherche en profondeur d'abord dans le graphe où une arête orientée va de x à y si $f(x) = y$, $f(y) = x$ ou $\mathtt{opp}(x) = y$, pour deux sommets quelconques x et y de $\{0, \ldots, n-1\}$. La commande (`nlist` n f x) retourne la liste des voisins de x dans ce graphe. La fonction auxiliaire `elimDup` élimine les doublons dans une liste. La recherche en profondeur d'abord est implémentée par la fonction `dfs` inspirée par la fonction du même nom de [Mathematical Components team, 2015]. La fonction `fold_left` est telle que `(fold_left f [x1;..;xk] a)` calcule `f (..(f (f a x1) x2)..) xk`.

Prouver la correction de cette implémentation par rapport à sa spécification présentée dans le listing 8.7 est une tâche difficile et est donc laissé en perspective.

```
Fixpoint dfs (g:nat → list nat) (n:nat) (v:list nat) (x:nat) :=
 if (in_dec eq_nat_dec x v) then
  v
 else
  match n with
```

```
    0 ⇒ v
  | S n' ⇒ fold_left (dfs g n') (g x) (x::v)
  end.

Definition is_transitive_funb n f := if
 eq_nat_dec n (length (dfs (nlist n f) n nil 0))
 then true else false.
```

Listing 8.13 – Fonction booléenne pour la transitivité.

La correction de `is_transitive_funb` peut cependant être validée, par exemple en comptant les premiers nombres de permutations transitives. En effet, rappelons qu'une carte enracinée est une classe d'équivalence de cartes étiquetées isomorphes par les isomorphismes enracinés (voir chapitre 2). Si $e > 0$, rappelons que le nombre $t(e)$ de permutations transitives de longueur $2e$ est égal au nombre de cartes enracinées multiplié par le nombre $2^{e-1}(e-1)!$ de cartes étiquetées locales isomorphes dans une carte enracinée.

Soit `d` $= 2e$ un entier naturel Coq pair et `l` une liste Coq de toutes les listes de permutation de longueur `d`. Le code Coq

```
Definition is_transitive_listb d l := is_transitive_funb d
   (list2fun l).
Eval compute in (length (filter (is_transitive_listb d) l).
```

calcule la longueur de la liste obtenue en filtrant les listes de permutation transitives, soit $t(e)$. La liste `l` est générée par une adaptation du BET en Prolog présenté dans la partie 8.1. Cette validation n'est faisable que pour $e = 0, 1, 2, 3$. Elle compte correctement $t(e) = 1, 2, 20, 592$ après avoir examiné $(2e)!$ permutations. Pour $e = 4$, la compilation Coq est à court de mémoire.

```
Definition isthmic_transb d1 (r1 : nat → nat)
                                d2 (r2 : nat → nat) :=
if even_odd_dec d1 then
 if even_odd_dec d2 then
  if is_permutb d1 r1 && is_transitive_funb d1 r1
     && is_permutb d2 r2 && is_transitive_funb d2 r2 then
   is_transitive_funb (S (S (d1 + d2))) (isthmic_fun d1 r1 d2 r2)
  else true
 else true
else true.
Definition non_isthmic_transb d (r : nat → nat) k (Hk : k ≤ d) :=
if even_odd_dec d then
 if is_permutb d r && is_transitive_funb d r then
  is_transitive_funb (S (S d)) (@non_isthmic_fun d r k k_le_d)
 else true
else true.
```

Listing 8.14 – Fonctions booléennes de test des théorèmes présentés dans le listing 8.12.

Après cette validation par comptage, nous utilisons la fonction booléenne `is_transitive_funb` pour tester les théorèmes présentés dans le listing 8.12.

Plus précisément, nous validons que tous les appels aux fonctions booléennes présentées dans le listing 8.14 sont valables pour toutes les permutations jusqu'à une certaine longueur.

L'opération isthmique combine insertion et somme directe. On pourrait penser qu'elle préserve la transitivité parce que ces deux opérations le font aussi. En réalité, c'est plus complexe. En particulier, l'opération de somme directe ne préserve pas la transitivité. Nous pouvons nous en apercevoir en revenant à sa définition. Mais cette préservation peut rapidement être invalidée par BET, sur une version exécutable du théorème suivant :

```
Theorem sum_transitive: ∀ d1 r1 d2 r2, even d1 → even d2 →
 is_permut d1 r1 → is_transitive_fun d1 r1 →
 is_permut d2 r2 → is_transitive_fun d2 r2 →
 is_transitive_fun (d1 + d2) (sum_fun d1 r1 d2 r2).
```

Le BET nous fournit les plus petits contre-exemples où `r1` et `r2` sont les fonctions `(list2fun [1;0])` encodant les transpositions échangeant 0 et 1.

Le test aléatoire nous permet également d'invalider cette conjecture et d'obtenir certains contre-exemples. Il n'existe pas de procédé pour produire des permutations transitives. Nous générons donc les permutations en tant que fonctions (comme précédemment) et nous filtrons celles qui sont transitives en utilisant le prédicat booléen `is_transitive_funb`. La commande `QuickCheck` suivante effectue cette tâche. Si `f1` (ou `f2`) n'est pas transitive, le cas de test est ignoré grâce au combinateur ⟹).

```
QuickCheck (forAll gen_even (fun d1 ⇒
 forAll (gen_permutl d1)  (fun l1 ⇒ let f1 := list2fun l1 in
  is_transitive_funb d1 f1 ⟹
   forAll gen_even (fun d2 ⇒
    forAll (gen_permutl d2) (fun l2 ⇒ let f2 := list2fun l2 in
     is_transitive_funb d2 f2 ⟹
      is_transitive_funb (d1 + d2) (sum_fun d1 f1 d2 f2)))))).
```

Cette commande peut générer les plus petits contre-exemples précédents, ou d'autres contre-exemples, tels que

```
2 [0;1] 4 [2;0;1;3] *** Failed! After 3 tests and 0 shrinks.
```

Dans ce dernier cas, la somme directe des deux permutations transitives représentées par les listes `[0;1] et [2;0;1;3]` est la permutation représentée par la liste `[0;1;4;2;3;5]` qui n'est pas transitive car les valeurs (brins) `0` et `3` (par exemple) ne sont pas connectées.

Chaque preuve de préservation de la transitivité se réduit à la préservation de la connectivité entre deux nombres (brins) x et y avec $x < y$. La preuve du théorème `isthmic_trans` décompose cette tâche en huit cas :

1. $y = d_1 + d_2 + 1$ (nouvelle racine) et $x = d_1 + d_2$ (opposé de la racine)
2. $y = d_1 + d_2 + 1$ et $d_1 \le x < d_1 + d_2$ $(x \in R_2)$
3. $y = d_1 + d_2 + 1$ et $0 \le x < d_1$ $(x \in R_1)$
4. $y = d_1 + d_2$ et $d_1 \le x < d_1 + d_2$ $(x \in R_2)$
5. $y = d_1 + d_2$ et $0 \le x < d_1$ $(x \in R_1)$
6. $d_1 \le y < d_1 + d_2$ $(y \in R_2)$ et $d_1 \le x < d_1 + d_2$ $(x \in R_2)$

7. $d_1 \leq y < d_1 + d_2$ $(y \in R_2)$ et $0 \leq x < d_1$ $(x \in R_1)$

8. $0 \leq y < d_1$ $(y \in R_1)$ et $0 \leq x < d_1$ $(x \in R_1)$

Les preuves de tous les cas sont similaires. Par exemple, dans le cas le plus complexe 7, la preuve construit un chemin entre x et y par concaténation d'un chemin du brin x jusqu'à la racine $d_1 - 1$ de R_1, d'un pas entre $d_1 - 1$ et la racine $d - 1 = d_1 + d_2 + 1$, d'un pas entre la racine $d - 1$ et son opposé $d - 2 = d_1 + d_2$ par l'involution sans point fixe `opp`, d'un pas de $d - 2$ jusqu'à la racine $d_2 - 1$ de R_2, renommée $d_2 - 1 + d_1$, et enfin d'un chemin de ce brin $d_2 - 1 + d_1$ jusqu'à y dans R_2 (dont les étiquettes ont été modifiées).

8.2.4 STATISTIQUES

L'étude de cas est composée de 40 définitions, 79 lemmes et 2 théorèmes, pour un total de 5580 lignes de code Coq. Parmi elles, environ 190 lignes sont dédiées à la validation. Ces lignes contiennent 23 définitions et 4 lemmes. Elles comprennent des versions booléennes de certaines définitions logiques utilisées à la fois par des tests aléatoires et du BET (par exemple la fonction booléenne `is_permutb`), leur preuves de correction correspondantes, et les générateurs requis par QuickChick. Le code Prolog pour le BET est composé de 44 lignes ajoutées à la bibliothèque de validation et de 897 lignes dont l'exécution génère des suites de tests pour l'étude de cas.

Toutes les validations par comptage et BET sont exécutées avec des listes de longueur au plus 4, en moins de 21 secondes sur un PC Intel Core i5-2400 3.10 GHz × 4 sous Linux Ubuntu 14.04 (le temps pour la génération des tests est négligeable). Les tests aléatoires QuickChick (10000 cas de test pour chaque étape de validation, sauf pour la conjecture fausse) sont générés et exécutés en moins de 54 secondes. Le test pour la validation de la transitivité est beaucoup moins efficace, parce qu'il procède par filtrage des permutations transitives parmi les permutations. Afin d'obtenir davantage de données de tests aléatoires que de données de tests par BET, nous avons effectué autant de tests réussis qu'il existe de permutations transitives de longueur 6, soit 432. Pour toutes les validations de la préservation de la transitivité (sauf pour la conjecture fausse), 432 cas de test sont générés aléatoirement et exécutés par QuickChick en moins de 71 secondes. Par comparaison, le temps de compilation Coq est d'environ 20 secondes.

8.3 SYNTHÈSE

Dans ce chapitre, nous avons formalisé en Coq les permutations et les cartes, ainsi que deux opérations de construction de cartes. Avant de prouver formellement les théorèmes sur ces opérations, nous les validons en utilisant la programmation logique (Prolog) pour effectuer du BET, et Coq/QuickChick pour effectuer du test aléatoire. C'est une étude expérimentale proposant une nouvelle méthodologie, qui permet d'envisager en perspective des projets encore plus ambitieux de formalisation de résultats combinatoires assistée par des outils de vérification.

La décomposition des cartes planaires présentée utilise l'opération basique de suppression d'une arête, dont on trouve une définition formelle (sur papier) dans [Lazarus, 2014]. Notons que ce travail a été facilité par la simplification de la formalisation des cartes, à travers la notion de carte locale. L'éventail de preuves envisageables en Coq est très large

du fait de l'interactivité. Néanmoins, une partie des preuves faites dans cette étude peut s'effectuer automatiquement. Poursuivant notre objectif d'automatisation des preuves, nous avons réalisé une adaptation de ce travail en C, décrite dans le chapitre 9.

9

GÉNÉRATION DE CARTES NON ÉTIQUETÉES

Nous avons présenté dans le chapitre 8 des opérations de construction de cartes, dont la correction a été formellement vérifiée en Coq. Ces opérations sont basées sur les deux opérations sur les permutations *insert* et *sum* étudiées dans le chapitre 4. Dans le présent chapitre, nous présentons une implémentation en C et une spécification ACSL des opérations de construction de cartes, afin d'automatiser certaines preuves effectuées auparavant de manière interactive en Coq.

De plus, nous validons sur ces fonctions C une propriété dite “de non isomorphisme (enraciné)”, qui nous permet de valider des théorèmes de construction de cartes enracinées (non étiquetées) et d'en dériver des générateurs exhaustifs bornés (testés) de cartes non étiquetées. Les preuves formelles de ces théorèmes sont laissées en perspectives.

Cette étude dépasse par ailleurs le cadre d'investigation du chapitre 8, puisqu'elle traite non seulement les cartes ordinaires de tout genre, mais aussi des cartes planaires (de genre 0).

Les contributions de ce chapitre sont :

1. Une implémentation en C des opérations de construction de cartes présentées dans le chapitre 8, ainsi que la preuve formelle automatique qu'elles préservent les permutations.
2. Un théorème de décomposition de cartes fondé sur ces opérations.
3. Un générateur exhaustif borné de cartes enracinées utilisant ces opérations.
4. Une implémentation en C d'opérations de construction de cartes planaires définies par W. Tutte [Tutte, 1968], ainsi que la preuve formelle automatique qu'elles préservent les permutations.
5. Un théorème de décomposition de cartes planaires fondé sur ces opérations.
6. Un générateur exhaustif borné de cartes planaires enracinées, utilisant ces opérations.
7. Une implémentation en C d'une variante des opérations de construction de cartes planaires, ainsi que la preuve formelle automatique qu'elles préservent les permutations.
8. Un théorème de décomposition de cartes planaires fondé sur cette variante.
9. Un générateur exhaustif borné de cartes planaires utilisant cette variante.

La partie 9.1 présente notre implémentation en C des opérations sur les cartes présentées dans le chapitre 8, une preuve formelle automatique qu'elles préservent les permutations, un théorème de décomposition de cartes fondé sur ces opérations, ainsi qu'un générateur exhaustif borné des cartes utilisant ces opérations. Les parties 9.2 et 9.3 présentent deux études similaires d'opérations de construction de cartes planaires, selon la même progression que dans la partie 9.1. Une synthèse et des perspectives sont ensuite proposées dans la partie 9.4.

Les fichiers sources de cette étude sont disponibles dans les répertoires `maps/rom/` et `maps/rpm/` de la bibliothèque `enum`.

9.1 Construction des cartes enracinées

Dans cette partie, nous implémentons en C les opérations de construction de cartes présentées dans le chapitre 8. Les parties 9.1.1 et 9.1.2 présentent respectivement une fonction C de construction des cartes isthmiques et non isthmiques. La partie 9.1.3 présente des preuves et des tests de propriétés de ces fonctions. Des tests complémentaires nous permettent d'énoncer un théorème de caractérisation complète des cartes enracinées, présenté dans la partie 9.1.4, et de proposer un générateur de cartes enracinées fondé sur cette caractérisation.

9.1.1 Cas isthmique

Le listing 9.1 présente une fonction C `isthmic`, munie de son contrat ACSL, implémentant l'opération c_I permettant de construire une carte locale isthmique M, à partir de deux cartes locales plus petites M_1 et M_2. Cette fonction est une implémentation sur des tableaux d'entiers. Elle correspond à la fonction Coq `isthmic_fun` décrite dans le chapitre 8 (listing 8.9).

La fonction `isthmic` prend en paramètre les trois tableaux r_1 (de longueur d_1), r_2 (de longueur d_2) et r (de longueur $d_1 + d_2 + 2$) stockant respectivement les rotations R_1, R_2 et R des cartes M_1, M_2 et M.

```
1  /*@ requires d1 ≥ 0 ∧ d2 ≥ 0;
2    @ requires \valid(r1+(0..d1-1)) ∧ \valid(r2+(0..d2-1)) ∧ \valid(r+(0..d1+d2+1));
3    @ requires \separated(r+(0..d1+d2+1),r1+(0..d1-1),r2+(0..d2-1));
4    @ requires is_perm(r1,d1) ∧ is_perm(r2,d2);
5    @ assigns r[0..d1+d2+1];
6    @ ensures is_perm(r,d1+d2+2); */
7  void isthmic(int r1[], int r2[], int r[], int d1, int d2) {
8   if (d1 == 0 ∧ d2 == 0) { insert(r1,0,r,0); insert_inplace(r,1,1); }
9   else if (d1 == 0) { insert(r2,d2-1,r,d2); insert_inplace(r,d2+1,d2+1); }
10  else if (d2 == 0) { insert(r1,d1,r,d1); insert_inplace(r,d1-1,d1+1); }
11  else {
12   sum(r1,r2,r,d1,d2);
13   insert_inplace(r,d1+d2-1,d1+d2);
14   insert_inplace(r,d1-1,d1+d2+1);
15  }
16 }
```

Listing 9.1 – Fonction `isthmic` annotée.

Dans le cas où l'une des deux cartes en entrée (ou les deux) est la carte vide, l'insertion des deux brins utilise uniquement les fonctions `insert` et `insert_inplace` (lignes 9-

11). Le premier brin est inséré grâce à la fonction insert qui construit partiellement la permutation R, et le deuxième brin est inséré grâce à la fonction insert_inplace (pour éviter l'utilisation d'une permutation supplémentaire) qui complète la construction de R. Dans le cas contraire, la somme directe de R_1 et R_2 est construite dans R grâce à la fonction sum, puis les brins sont insérés grâce à deux appels successifs de la fonction insert_inplace (lignes 12-14).

La fonction isthmic est munie d'un contrat ACSL (lignes 1-6) comportant la postcondition is_perm(r,d1+d2+2) affirmant que le tableau r en sortie est une permutation. La preuve de ce contrat est décrite dans la partie 9.1.3.

9.1.2 CAS NON ISTHMIQUE

Le listing 9.2 présente une fonction C non_isthmic, munie de son contrat ACSL, implémentant l'opération c_N^k (présentée dans le chapitre 8) permettant de construire une carte locale non isthmique M à partir d'une carte locale M_1, par ajout d'une arête. Cette fonction s'applique à des tableaux d'entiers. Elle correspond à la fonction Coq non_isthmic_fun décrite dans le chapitre 8 (listing 8.10).

La fonction non_isthmic prend en paramètres deux tableaux r_1 (de longueur d_1) et r (de longueur d_1+2) stockant respectivement les rotations R_1 et R des cartes M_1 et M. La permutation résultante R est obtenue par insertion de la nouvelle racine $d+1$ et de son opposé d dans R. La nouvelle arête est ajoutée grâce aux appels successifs des fonctions insert et insert_inplace. Si M est la carte vide, la fonction non_isthmic retourne 0 après l'insertion de cette arête, signifiant qu'il n'existe pas d'autre manière de l'ajouter dans la carte M (ligne 7). Sinon, il existe $d+1$ manières d'ajouter cette nouvelle arête, distinguées par les valeurs du paramètre k entre 0 et d. Cet ajout s'effectue ligne 8, puis la fonction non_isthmic retourne $k+1$ (ligne 10), indiquant la prochaine manière d'ajouter cette arête, sauf si k est égal à d_1. La fonction retourne alors 0 (ligne 9) car toutes les manières d'insérer cette arête ont été épuisées.

On munit la fonction non_isthmic d'un contrat ACSL (lignes 1-5) comportant la postcondition is_perm(r,d1+2) affirmant que le tableau r en sortie est une permutation. La preuve de ce contrat est décrite dans la partie 9.1.3.

```
1  /*@ requires 0 ≤ k ≤ d1 ∧ \valid(r1+(0..d1-1)) ∧ \valid(r+(0..d1+1));
2    @ requires \separated(r+(0..d1+1),r1+(0..d1-1));
3    @ requires is_perm(r1,d1);
4    @ assigns r[0..d1+1];
5    @ ensures is_perm(r,d1+2); */
6  int non_isthmic(int r1[], int k, int r[], int d1) {
7   if (d1 == 0) { insert(r1,0,r,0); insert_inplace(r,0,1); return 0; }
8   insert(r1,d1-1,r,d1); insert_inplace(r,k,d1+1);
9   if (k == d1) return 0;
10  return k+1;
11 }
```

Listing 9.2 – Fonction non_isthmic.

9.1.3 PREUVES ET TESTS

Les contrats des fonctions isthmic et non_isthmic (présentés dans les listings 9.1 et 9.2) sont automatiquement vérifiés par le greffon WP de Frama-C, sans annotation ACSL

supplémentaire. En effet, leur correction est une conséquence élémentaire de celle des contrats des fonctions implémentant les opérations d'insertion et de somme directe sur les permutations. Les fonctions `isthmic` et `non_isthmic`, qui utilisent ces fonctions d'insertion et de somme directe, héritent ainsi de cette propriété.

Pour spécifier que la permutation R construite par les opérations c_I et c_N^k représente bien une carte locale, nous devrions spécifier sa transitivité en ACSL. Dans le cadre de ce mémoire, nous écartons cette piste, car nous avons vu dans le chapitre 5 que cette spécification pouvait compromettre l'automaticité de la vérification. Nous validons donc la transitivité par BET. Par abus de langage, même avant d'avoir validé cette propriété, nous avons désigné et désignerons encore dans la suite par “carte locale” tout tableau construit par les fonctions `isthmic` et `non_isthmic`.

```
1 void rom_atlas(int ***rom, int emax) {
2  int e,e1,e2,k;
3  long c1,c2,i,i1,i2;
4
5  for (e = 1; e ≤ emax; e++) {
6   i = 0;
7   for (e1 = 0; e1 < e; e1++) { // isthmic case
8    e2 = e-1-e1;
9    c1 = count698(e1+1);
10   c2 = count698(e2+1);
11   for (i1 = 0; i1 < c1; i1++) {
12    for (i2 = 0; i2 < c2; i2++) {
13     isthmic(rom[e1][i1],rom[e2][i2],rom[e][i],2*e1,2*e2);
14     if (! b_trans_local(rom[e][i],2*e)) exit(1);
15     is_new_local(rom,e,i);
16     i++;
17    }
18   }
19  }
20  c1 = count698(e);
21  for (i1 = 0; i1 < c1; i1++) { // non-isthmic case
22   k = 0;
23   do {
24    k = non_isthmic(rom[e-1][i1],k,rom[e][i],2*(e-1));
25    if (! b_trans_local(rom[e][i],2*e)) exit(1);
26    is_new_local(rom,e,i);
27    i++;
28   } while (k ≠ 0);
29  }
30  if (i ≠ count698(e+1)) exit(1);
31 }
32 }
```

Listing 9.3 – Fonction `rom_atlas`.

La fonction `rom_atlas` reproduite dans le listing 9.3 construit des cartes locales à e arêtes (pour $e > 0$) à partir de cartes locales possédant entre 0 et $e - 1$ arêtes, grâce aux fonctions `isthmic` et `non_isthmic`. Les cartes générées par `rom_atlas` sont stockées dans un tableau d'entiers rom à trois dimensions (supposé pré-alloué en mémoire). Chaque ligne $rom[e]$ de ce tableau est destinée à stocker des cartes locales à e arêtes (avec $e \leq emax$) produites par les fonctions `isthmic` et `non_isthmic`. La taille de la deuxième dimension du tableau `rom` est fixée au nombre de cartes non étiquetées à e arêtes, égal à $A000698(e + 1)$ et calculé par `count698(e+1)`. Ce choix sera justifié dans la partie 9.1.4. Pour tout entier i dans les bornes du tableau $rom[e]$, le tableau $rom[e][i]$ stocke une carte locale, représentée par sa rotation, qui est une permutation de taille $2e$.

Avant appel de la fonction `rom_atlas`, la première ligne `rom[0]` du tableau `rom` doit être construite, de taille 1, réduite au tableau `rom[0][0]` de longueur `0`, qui représente la carte vide. Pour `e` non nul, la fonction `rom_atlas` remplit la ligne `rom[e]` avec des

cartes locales de taille `e` construites à partir de cartes locales de taille inférieure, stockées dans les lignes précédentes, en deux étapes successives :

— Cas isthmique (lignes 7-19) : pour tous les nombres e_1 d'arêtes jusqu'à e exclu, la fonction `isthmic` est appliquée à tous les couples de cartes locales stockées, de tailles e_1 et e_2 dont la somme est égale à $e - 1$.

— Cas non isthmique (lignes 20-29) : la fonction `non_isthmic` est appliquée à toutes les cartes locales stockées à $e - 1$ arêtes, avec toutes les valeurs possibles du paramètre k entre 0 et $e - 1$ inclus. Lorsque la fonction `non_isthmic` retourne la valeur 0, toutes les possibilités d'insérer la nouvelle arête ont été épuisées, et la génération se termine pour les cartes de taille e.

Immédiatement après la génération et le stockage de chaque permutation, nous pouvons valider qu'elle est transitive. Il suffit pour cela d'utiliser la fonction booléenne `b_trans_local` qui caractérise la transitivité d'une permutation encodant une carte locale (lignes 14 et 25). Dans le chapitre 5, cette fonction joue un rôle de générateur de cartes locales (par filtrage). Elle sert ici d'accepteur de cartes locales.

D'autre part, nous validons grâce à la fonction `is_new` (lignes 15 et 26) que chaque permutation générée par les fonctions `isthmic` et `non_isthmic` est différente de toutes celles de même taille générées précédemment et stockées dans le tableau rom, par comparaison de leurs valeurs. Enfin, nous effectuons pour chaque taille e une comparaison entre le nombre de cartes générées et le nombre de cartes enracinées de taille e, afin de valider que chaque ligne du tableau rom est complètement remplie (ligne 30).

Ces tests valident que les opérations c_I et c_N^k ne produisent que des cartes locales, et que des applications différentes de ces opérations produisent des cartes différentes. Dans la partie suivante, nous validons de plus que ces cartes ne sont pas équivalentes entre elles, par un isomorphisme enraciné.

9.1.4 Décomposition des cartes enracinées

La partie 3 de [Walsh et al., 1972a] décrit (informellement) une décomposition des cartes enracinées indépendamment de leur genre, par effacement de leur arête racine, et en dérive directement la relation de recurrence suivante, pour les nombres $F_{b,p}$ de cartes enracinées avec $b + p$ arêtes et $p + 1$ sommets : $F_{0,0} = 1$ et

$$F_{b,p} = \sum_{j=0}^{p-1} \sum_{k=0}^{b} F_{k,j} F_{b-k,p-j-1} + [2(b+p) - 1] F_{b-1,p} \tag{9.1}$$

si b ou p est non nul. Nos opérations isthmique et non isthmique coïncident avec cette décomposition. Nous pouvons donc reformuler comme suit la décomposition de [Walsh et al., 1972a].

Théorème 9 ([Walsh et al., 1972a]). *Soit M une carte enracinée possédant $e(M)$ arêtes. M vérifie alors exactement l'un des cas suivants :*

1. *M est la carte vide et $e(M) = 0$.*
2. *Cas isthmique : $M = c_I(M_1, M_2)$ pour deux cartes M_1 et M_2 telles que $e(M) = 1 + e(M_1) + e(M_2)$.*
3. *Cas non isthmique : $M = c_N^k(M_1)$ pour une carte M_1 telle que $e(M) = 1 + e(M_1)$ et un entier k compris entre 0 et $2e(M_1)$ inclus.*

Bien que ce théorème ne soit pas explicite dans [Walsh et al., 1972a], nous attribuons sa paternité à Walsh et Lehman.

Une preuve formelle de ce théorème est laissée en perspective. Auparavant, nous le validons ici en complétant les preuves et tests de la partie 9.1.3 par un BET de la propriété de non-isomorphisme (enraciné) selon laquelle la fonction `rom_atlas` construit exactement une carte locale par classe d'isomorphisme enraciné, c'est-à-dire exactement un représentant de chaque carte combinatoire enracinée non étiquetée.

Cette validation s'effectue en remplaçant la fonction `is_new` par une fonction `is_not_iso` dans les lignes 15 et 26 du listing 9.3.

Pour chaque carte locale (d'une taille donnée) produite, cette fonction génère toutes les cartes locales qui lui sont isomorphes par un isomorphisme enraciné préservant l'involution sans point fixe locale, ici notée L, et vérifie qu'aucune d'entre elles n'a été stockée précédemment dans le tableau rom.

A cette fin, nous avons conçu un générateur séquentiel de permutations sur $\{0, \ldots, 2e-1\}$ préservant la racine $(2e-1)$ et l'involution L, par filtrage parmi les permutations. Ce n'est pas d'une efficacité optimale, mais c'est très facile à implémenter et nous verrons que cela produit déjà un grand nombre de tests dans un délai raisonnable. Ce générateur produit $2^{e-1}(e-1)!$ permutations θ de taille $2e$ telles que $\theta(2e-1) = 2e-1$, $L = \theta^{-1}\, L\, \theta$, et $\theta(2e-2) = 2e-2$, conséquence des deux égalités précédentes. En conjuguant chaque carte locale générée (par la fonction `rom_atlas`) avec chaque renommage produit par ce générateur, la fonction `is_not_iso` génère les $2^{e-1}(e-1)!-1$ autres cartes locales qui lui sont isomorphes, puis valide leur absence dans le tableau rom à un indice inférieur.

Propriété	Nombre d'arêtes	Nombre de cartes générées	Nombre de cartes stockées	Durée de calcul (s)
Transitivité	8	31 579 825	31 579 825	9
Unicité	6	119 365	119 365	44
Non isomorphisme	5	329 601 959	8 955	79

TABLE 9.1 – Statistiques de test de la génération de cartes enracinées.

Des statistiques de test de la génération de cartes enracinées sont présentées dans la table 9.1. La première colonne indique la propriété validée, la deuxième donne le nombre maximal d'arêtes pour lequel chaque validation a été faite, les troisième et quatrième colonnes donnent respectivement le nombre total de cartes générées (avant filtrage) et stockées, tandis que la dernière colonne indique la durée de calcul correspondante (en secondes).

Ces validations nous permettent d'affirmer que la fonction `rom_atlas` est un générateur exhaustif borné de cartes enracinées (non étiquetées).

9.2 CONSTRUCTION DES CARTES PLANAIRES ENRACINÉES

Nous adaptons l'étude de la partie 9.1 aux cartes locales **planaires** (c.a.d. de genre 0).

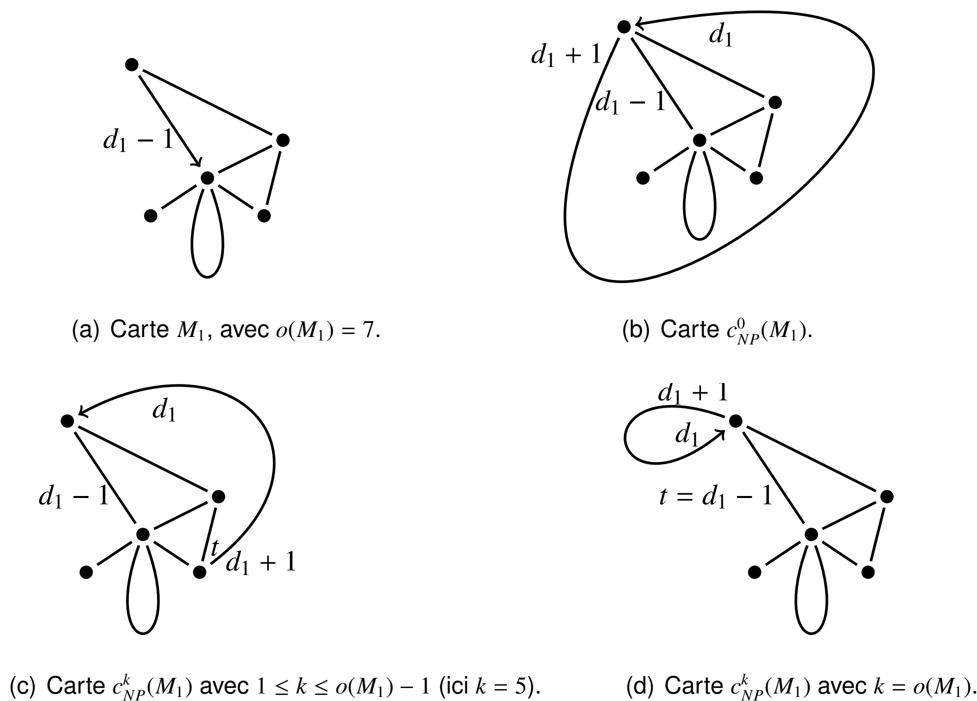
En 1968, W. Tutte a établi une décomposition inductive des cartes planaires enracinées (non étiquetées) [Tutte, 1968], en distinguant le cas isthmique et le cas non isthmique. Le cas isthmique n'est que la restriction au genre 0 du cas isthmique pour les cartes de genre quelconque. Pour les cartes planaires, le cas non isthmique diffère plus fondamen-

talement du cas non isthmique des cartes de genre quelconque. En effet, pour que la planarité soit préservée, les possibilités d'ajout d'arête sont plus limitées. Ces opérations permettent la génération des cartes locales planaires, que nous désignerons par la suite par PLM, pour *Planar Local Map*.

Les parties 9.2.1 et 9.2.2 exposent respectivement l'opération de construction de cartes locales planaires dans le cas non isthmique, et son implémentation. La partie 9.2.3 présentent des preuves et des tests de propriétés des deux opérations de construction de cartes locales planaires. Dans la partie 9.2.4, des tests complémentaires nous permettent d'énoncer un théorème de caractérisation complète des cartes planaires enracinées, et de proposer un générateur de cartes planaires enracinées fondé sur cette caractérisation.

9.2.1 Cas non isthmique

Dans le cas isthmique, l'opération c_I de construction de cartes locales présentée dans le chapitre 8 additionne les genres des deux cartes qu'elle assemble, et donc permet d'obtenir des cartes planaires à partir de cartes planaires. Dans le cas non isthmique,



(a) Carte M_1, avec $o(M_1) = 7$.

(b) Carte $c^0_{NP}(M_1)$.

(c) Carte $c^k_{NP}(M_1)$ avec $1 \leq k \leq o(M_1) - 1$ (ici $k = 5$).

(d) Carte $c^k_{NP}(M_1)$ avec $k = o(M_1)$.

FIGURE 9.1 – Illustration de l'opération c^k_{NP} et de la fonction `non_isthmic_rpm`.

l'opération de construction de cartes locales diffère lorsqu'on se restreint aux cartes planaires. Rappelons que cette opération de construction à partir d'une carte M_1 dépend d'un paramètre entier k, qui détermine le brin de M_1 qui précède immédiatement l'un des deux brins ajoutés à M_1. Pour les cartes de tout genre, l'entier k est compris entre 0 et le nombre de brins de M_1. Comme expliqué dans la partie 8.2.2.2 du chapitre 8, lorsque le brin ajouté n'est pas incident à la même face que la face incidente à la racine de M_1, la nouvelle arête ne peut être ajoutée que grâce à un trou perforant la surface. Pour éviter ce cas et ne construire que des cartes planaires, l'entier k doit être compris entre 0 et le degré $o(M_1)$ de la face extérieure de M_1. Ce degré et sa notation sont définis dans la

partie 2.1.4.4 du chapitre 2.

Soit M_1 une PLM de face extérieure de degré $o(M_1)$. Soit k un entier compris entre 0 et $o(M_1)$ inclus. L'opération de construction de Tutte, dans le cas non isthmique, est nommée c^k_{NP}. Elle est illustrée par la figure 9.1. La sous-figure 9.1(a) présente une carte M_1 dont la face extérieure est de degré $o(M_1) = 7$. Les trois autres sous-figures présentent trois cartes construites à partir de M_1 par application de l'opération c^k_{NP}, pour trois valeurs différentes de k. Les cartes des sous-figures 9.1(b) et 9.1(d) sont obtenues respectivement pour la valeur minimale 0 et maximale $o(M_1) = 7$ de l'entier k. La carte de la sous-figure 9.1(c) est obtenue pour une valeur intermédiaire de k, égale à 5 dans cet exemple.

La PLM $c^k_{NP}(M_1)$ est construite par ajout d'une arête dans M_1, dont l'un des brins est inséré de manière à être incident à la face extérieure de M_1 à une position indiquée par l'entier k. Suite à cet ajout, la face extérieure de la carte $M = c^k_{NP}(M_1)$ est de degré $o(M) = k + 1$. L'action de cette opération est détaillée dans la partie 9.2.2.

9.2.2 Implémentation

L'opération c^k_{NP} du cas non isthmique de la construction de Tutte est implémentée par la fonction `non_isthmic_rpm` reproduite dans le listing 9.4.

```
1 int non_isthmic_rpm(int r1[], int k, int r[], int e1) {
2  int i,t,d1 = 2*e1;
3
4  if (e1 == 0) {
5   insert(r1,0,r,0);
6   insert_inplace(r,0,1);
7   return 0;
8  }
9  insert(r1,d1-1,r,d1);
10 if (k == 0) {
11  insert_inplace(r,d1-1,d1+1);
12  return 1;
13 }
14 for (t = d1-1, i = 0; i < k; i++) t = r1[invol(t)];
15 if (t == d1-1) {
16  insert_inplace(r,d1,d1+1);
17  return 0;
18 }
19 insert_inplace(r,t,d1+1);
20 return k+1;
21 }
```

Listing 9.4 – Fonction `non_isthmic_rpm`.

Cette construction ajoute une arête à une PLM M_1 ayant e_1 arêtes (représentée par une permutation R_1 de taille $d_1 = 2e_1$ et encodée par le tableau r_1 de longueur d_1) pour obtenir une PLM M ayant e_1+1 arêtes (représentée par une permutation R de taille $d = d_1+2$). La fonction `non_isthmic_rpm` est similaire à la fonction `non_isthmic`, présentée dans la partie 9.1.2. Elle ajoute deux brins $d_1 + 1$ et d_1 dans les cycles de R_1 grâce à des appels successifs aux fonctions `insert` et `insert_inplace`. Si M_1 est la carte vide, l'arête ajoutée est une boucle et la fonction retourne 0 (lignes 4 à 8). Dans tous les autres cas, le brin d_1 est inséré avant la racine $d_1 - 1$ de M_1 (ligne 9). L'insertion du brin $d_1 + 1$ diffère ensuite selon la valeur de k.

Si $k = 0$, l'arête racine de $M = c^0_{NP}(M_1)$ forme une boucle dont la face extérieure est de degré $o(M) = 1$, comme illustré dans la sous-figure 9.1(b). Ce cas est traité lignes 10 à 13

du listing 9.4. Le brin $d_1 + 1$ est inséré avant le brin $d_1 - 1$ de telle sorte que l'arête ajoutée forme une boucle et que $o(M) = 1$. La fonction retourne alors la valeur suivante de k, soit 1.

Le cas général complémentaire $k > 0$, illustré par les figures 9.1(c) et 9.1(d), nécessite de déterminer où insérer le nouveau brin racine d_1+1 de M. Soit $o(M_1) \geq 1$ le degré de la face extérieure de M_1. Il existe $o(M_1)+1$ manières d'ajouter la nouvelle arête. Le sommet initial de l'arête ajoutée dépend de la valeur du paramètre k. Ce paramètre indique le nombre de brins à parcourir le long de la face extérieure de M_1, à partir du sommet incident à sa racine et dans la direction de sa racine, pour atteindre le sommet incident à la nouvelle racine. Le brin t avant lequel insérer le brin $d_1 + 1$ est ainsi déterminé par k itérations de la boucle ligne 14, qui parcourt k brins le long de la face extérieure de M_1. Si $k = o(M_1)$ (voir figure 9.1(d)), le brin t est $d_1 - 1$ (ligne 15), l'arête ajoutée forme une boucle vide (c.a.d. telle que le degré $i(M)$ de la face incidente à l'opposé de la nouvelle racine $i(M)$ soit égal à 1), et la fonction retourne 0 pour signifier que la valeur maximale de k est atteinte. Dans les autres cas, illustrés par la figure 9.1(c), l'arête ajoutée n'est pas une boucle et la fonction retourne la valeur suivante de k (ligne 20).

9.2.3 Preuves et tests

Nous ne traitons par preuve que la propriété de préservation des permutations. Le contrat de la fonction `non_isthmic_rpm`, montré dans le listing 9.5, est similaire à celui de la fonction `non_isthmic`.

```
1 /*@ requires 0 ≤ k ≤ 2*e1 ∧ \valid(r1+(0..2*e1-1)) ∧ \valid(r+(0..2*e1+1));
2   @ requires \separated(r+(0..2*e1+1),r1+(0..2*e1-1));
3   @ requires is_perm(r1,2*e1);
4   @ assigns r[0..2*e1+1];
5   @ ensures is_perm(r,2*(e1+1)); */
6 int non_isthmic_rpm(int r1[], int k, int r[], int e1);
```

Listing 9.5 – contrat de la fonction `non_isthmic_rpm`.

Pour le prouver automatiquement, il suffit d'ajouter l'invariant de boucle

```
loop invariant 0 <= i <= k && 0 <= t <= d1-1;
```

dans les annotations de la boucle `for` entre les lignes 13 et 14 du listing 9.4.

En complément, les tableaux générés par les fonctions `isthmic` et `non_isthmic_rpm` sont testés selon la méthodologie détaillée dans la partie 9.1.3.

Grâce à la fonction `rpm_atlas` reproduite dans le listing 9.6, nous construisons un atlas de tableaux générés par les fonctions `isthmic` et `non_isthmic_rpm`. La fonction `rpm_atlas` est similaire à la fonction `rom_atlas`. Les cartes locales générées par `rpm_atlas` sont stockées dans un tableau d'entiers rpm à trois dimensions (supposé pré-alloué en mémoire). Chaque ligne $rpm[e]$ de ce tableau est destinée à stocker des cartes planaires locales à e arêtes (avec $e \leq emax$) produites par les fonctions `isthmic` et `non_isthmic_rpm`. Ici, la taille de la deuxième dimension du tableau `rpm` est fixée au nombre de cartes planaires enracinées égal à $A000168(e)$ et implémentée par la fonction `count168`. Dans la fonction `rpm_atlas`, le traitement des cas isthmique et non isthmique est le même que dans la fonction `rom_atlas`. Dans le cas non isthmique, la valeur de retour k de la fonction `non_isthmic_rpm` indique s'il reste des possibilités d'insérer la nouvelle arête dans une carte de taille $e-1$ de telle sorte que la nouvelle carte

produite de taille e soit planaire. Si la fonction retourne 0, ces possibilités sont épuisées et la génération des cartes de taille e se termine.

```
1  void rpm_atlas(int ***rpm, int emax) {
2   int d,e,e1,e2,k;
3   long i,c1,c2,i1,i2;
4
5   for (e = 1; e ≤ emax; e++) {
6    d = 2*e;           // number of darts
7    i = 0;
8    for (e1 = 0; e1 < e; e1++) { // isthmic case
9     e2 = e-1-e1;
10    c1 = count168(e1);
11    c2 = count168(e2);
12    for (i1 = 0; i1 < c1; i1++) {
13     for (i2 = 0; i2 < c2; i2++) {
14      isthmic(rpm[e1][i1],rpm[e2][i2],rpm[e][i],2*e1,2*e2);
15      if (! b_trans_local(rpm[e][i],d)) exit(1);
16      is_new_local(rpm,e,i);
17      if (! b_planar_local(rpm[e][i],d)) exit(1);
18      i++;
19     }
20    }
21   }
22   c1 = count168(e-1);
23   for (i1 = 0; i1 < c1; i1++) { // non-isthmic case
24    k = 0;
25    do {
26     k = non_isthmic_rpm(rpm[e-1][i1],k,rpm[e][i],e-1);
27     if (! b_trans_local(rpm[e][i],d)) exit(1);
28     is_new_local(rpm,e,i);
29     if (! b_planar_local(rpm[e][i],d)) exit(1);
30     i++;
31    } while (k ≠ 0);
32   }
33   if (i ≠ count168(e)) exit(1);
34  }
35 }
```

Listing 9.6 – Fonction `rpm_atlas`.

Pour chaque tableau généré, nous validons qu'il encode la permutation R d'une carte locale, et que cette carte est différente de toutes celles déjà générées (lignes 15-16 et 27-28). De plus, nous effectuons pour chaque taille e une comparaison entre le nombre de cartes générées et `count168(e)`, afin de valider que chaque ligne du tableau rpm est complètement remplie (ligne 33).

Par rapport aux tests effectués dans la partie 9.1.3, nous ajoutons ici un test de la planarité des cartes locales générées, grâce à la fonction `b_planar_local` (lignes 17 et 29), dont la conception est détaillée ci-après. Nous avons vu dans le chapitre 2 que le nombre de cycles de la permutation R d'une carte indique le nombre de sommets de cette carte. De même, le nombre de cycles de la permutation LR (composée de L et R, voir la partie 2.1.3 du chapitre 2) d'une carte indique son nombre de faces. Or, pour une carte possédant V sommets, E arêtes et F faces, la formule d'Euler $V - E + F = 2$ caractérise sa planarité. Afin d'utiliser cette caractérisation, nous construisons la permutation des faces par un appel à la fonction `composition` présentée dans le chapitre 4. Nous utilisons ensuite une fonction calculant le nombre de cycles d'une permutation pour obtenir le nombre de sommets et de faces de la carte représentée par la permutation R, et nous pouvons alors vérifier par une application de la formule d'Euler que chaque carte locale générée est planaire.

9.2.4 Décomposition des cartes planaires enracinées

Le théorème suivant explicite une caractérisation complète des cartes planaires enracinées selon leur nombre d'arêtes. Cette décomposition a été implicitement utilisée par Tutte pour énumérer ces cartes [Tutte, 1968].

Théorème 10 ([Tutte, 1968])**.** *Soit M une carte locale planaire ayant $e(M)$ arêtes et de degré de face extérieure $o(M)$. M vérifie alors exactement l'un des cas suivants :*

1. *M est la carte vide et $e(M) = o(M) = 0$.*
2. *$M = c_I(M_1, M_2)$ pour deux cartes M_1 et M_2 telles que $e(M) = 1 + e(M_1) + e(M_2)$ et $o(M) = 2 + o(M_1) + o(M_2)$.*
3. *$M = c^k_{NP}(M_1)$ pour une carte M_1 et un entier k compris entre 0 et $o(M_1)$ inclus tels que $e(M) = 1 + e(M_1)$ et $o(M) = k + 1$.*

Nous validons ce théorème par énumération, en montrant que la fonction `rpm_atlas` construit exactement une carte locale planaire par classe d'isomorphisme enraciné, c'est-à-dire exactement un représentant de chaque carte combinatoire planaire enracinée non étiquetée.

De manière analogue à la validation du théorème 9, cette validation s'effectue en remplaçant la fonction `is_new_local` par une fonction `is_not_iso_local` dans les lignes 14 et 25 du listing 9.6.

Pour chaque carte planaire locale (d'une taille donnée) produite, cette fonction génère toutes les cartes planaires locales qui lui sont isomorphes par un isomorphisme enraciné préservant l'involution sans point fixe locale L (détaillé dans la partie 9.1.4) et vérifie qu'aucune d'entre elles n'a été stockée précédemment dans le tableau rpm.

Validation	Nombre d'arêtes	Nombre de cartes générées	Nombre de cartes stockées	Durée de calcul (s)
Transitivité	9	19 512 128	19 512 128	6
Planarité	9	19 512 128	19 512 128	9
Unicité	7	235 910	235 910	171
Non isomorphisme	5	117 846 597	3 360	13

TABLE 9.2 – Statistiques de test de la génération de cartes planaires enracinées avec les opérations `isthmic` et `non_isthmic_rpm`.

Des statistiques de test de la génération de cartes planaires enracinées sont présentées dans la table 9.2. La première colonne indique la propriété validée, la deuxième donne le nombre maximal d'arêtes pour lequel chaque validation a été faite, les troisième et quatrième colonnes donnent respectivement le nombre total de cartes générées et stockées, tandis que la dernière colonne indique la durée de calcul correspondante (en secondes).

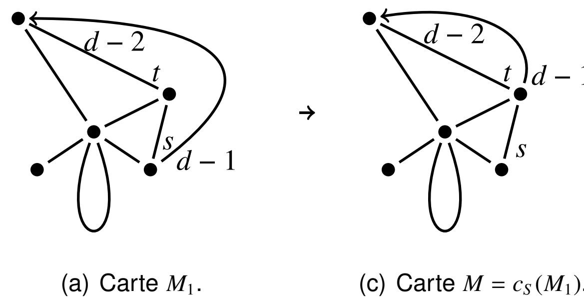
Ces validations nous permettent d'affirmer que la fonction `rpm_atlas` est un générateur exhaustif borné de cartes planaires enracinées (non étiquetées).

9.3 Variante de construction des cartes planaires enracinées

Dans le cas non isthmique, l'opération de construction c^k_{NP} est paramétrée par un entier k compris entre 0 et le degré $o(M_1)$ de la face extérieure de M_1, indiquant où insérer l'ori-

gine de la nouvelle arête. Pour éviter d'avoir à calculer cet emplacement à l'aide d'une boucle, nous présentons dans cette partie une variante de cette construction à l'aide de deux opérations. Cette variante consiste à n'ajouter une arête que dans l'un des $o(M_1)+1$ cas de construction. Dans les autres cas, on ne fait que décaler l'origine de cette arête. Ces opérations sont décrites dans la partie 9.3.1. L'implémentation en C de ces opérations est présentée dans la partie 9.3.2. Des preuves et des tests de propriétés de ces opérations sont ensuite décrits dans la partie 9.3.3. Dans la partie 9.3.4 des tests complémentaires nous permettent d'énoncer une caractérisation complète des cartes planaires enracinées, et de proposer un second générateur de cartes planaires enracinées.

9.3.1 Cas non isthmique



(a) Carte M_1. (c) Carte $M = c_S(M_1)$.

FIGURE 9.2 – Exemple de décalage de la racine par l'opération c_S.

La variante du cas non isthmique se décompose en deux opérations.

Une première opération, nommée c_L (L pour Loop), ajoute une boucle à une carte M_1 pour obtenir une carte M. Cette opération correspond au cas $k = 0$ de l'opération c^k_{NP}, c.a.d. $c_L = c^0_{NP}$. Par exemple, cette opération produit la carte présentée dans la figure 9.1(b) à partir de la carte présentée dans la figure 9.1(a).

Une deuxième opération, nommée c_S (S pour Shift), décale d'un brin la racine d'une carte M_1 le long de sa face extérieure pour obtenir une carte M. Cette opération n'ajoute pas d'arête à la carte M_1. Son action est illustrée par la figure 9.2. La carte $M = c_S(M_1)$ est construite par le processus suivant : partant du sommet incident à la racine de M_1, parcourir un brin le long de sa face extérieure pour atteindre le prochain sommet, et décaler la racine pour que ce sommet soit son sommet incident. En particulier, l'opération de décalage de racine a pour effet de décrémenter le degré $i(M_1)$ de la face intérieure de la carte de 1. Notons qu'une condition pour appliquer c_S est que M_1 soit une carte non isthmique avec une face intérieure de degré $i(M_1) \geq 2$.

9.3.2 Implémentation

L'introduction de la boucle par l'opération c_L est implementée par la fonction C `loop` reproduite dans le listing 9.7 lignes 1-5. Son code correspond au cas $k = 0$ de la fonction `non_isthmic_rpm` présentée dans la partie 9.2.2.

L'opération de décalage de racine c_S est implémentée par la fonction `shift` présentée dans le listing 9.7 lignes 7-14. Cette fonction prend en entrée une PLM M_1 représentée par la permutation R_1 et encodée par un tableau r_1 de longueur $d = 2e$ (avec $e > 0$),

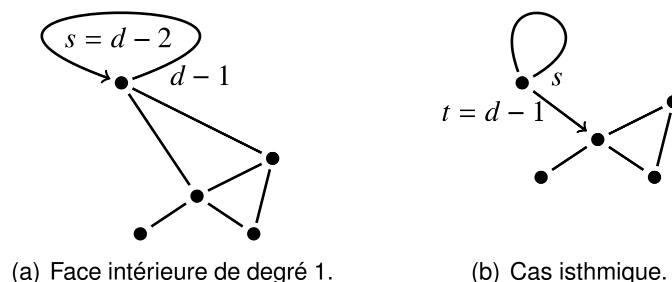
et construit une nouvelle PLM M représentée par la permutation R et encodée par un tableau r de même longueur que r_1.

```
1 void loop(int r1[], int r[], int e1) {
2  int d1 = 2*e1;
3  if (e1 == 0) { insert(r1,0,r,0); insert_inplace(r,0,1); }
4  else { insert(r1,d1-1,r,d1); insert_inplace(r,d1-1,d1+1); }
5 }
6
7 int shift(int r1[], int r[], int e) {
8  int s = r1[d-1], t = r1[invol(s)], d = 2*e;
9  if (s ≠ d-2 ∧ t ≠ d-1) {
10   remove_max(r1,d,r); insert_inplace(r,t,d-1);
11   if (t ≠ d-2) return 1;
12  }
13  return 0;
14 }
```

Listing 9.7 – Fonctions d'ajout d'une boucle et de décalage de la racine en C.

Soit $s = R_1(d-1)$ le brin qui suit immédiatement la racine $d-1$ de M_1 autour de son sommet incident et $t = L_1R_1(s)$ le brin qui suit s le long de la face intérieure de M_1 (voir un exemple dans la figure 9.2).

Le décalage de la racine $d-1$ ne peut pas s'effectuer si la face intérieure de M_1 est de degré inférieur à 2, c.a.d. si $s = d-2$. Ce cas est illustré par la figure 9.3(a). Dans le cas contraire, ce décalage s'effectue grâce aux instructions ligne 10, en ôtant la racine de son cycle de la permutation R_1 (par un appel à la fonction `remove_max` présentée dans le chapitre 4), puis en l'insérant avant t dans son cycle pour obtenir la permutation R (par un appel à la fonction `insert_inplace`). La fonction retourne alors 1 (ligne 11) si un décalage supplémentaire de la racine est possible (c.a.d. si $t \neq d-2$), et 0 sinon (ligne 13).



(a) Face intérieure de degré 1.

(b) Cas isthmique.

FIGURE 9.3 – Illustration des cas où la fonction `shift` retourne 0.

Lors de la construction, cette fonction ne s'applique qu'à des cartes non isthmiques (dont l'arête portant la racine n'est pas un isthme). Cependant, pour simplifier la spécification de cette fonction, nous envisageons tous les cas d'utilisation, notamment ceux où l'arête portant la racine est un isthme. La fonction `shift` s'applique aussi dans ces cas, sauf dans le cas particulier où $t = d-1$, illustré par la figure 9.3(b). C'est pourquoi la fonction `shift` n'effectue pas le décalage de la racine si $s = d-2$ ou $t = d-1$ (condition ligne 9), et retourne alors 0 sans modifier le tableau r.

9.3.3 Preuves et tests

Le code de la fonction `loop` étant constitué d'une partie du code de la fonction `non_-isthmic_rpm` (présentée dans la partie 9.2.2), son contrat et sa preuve sont similaires (sans paramètre k).

Le contrat ACSL de la fonction `shift`, présenté dans le listing 9.8, permet de prouver formellement qu'elle préserve les permutations, automatiquement.

```
1 /*@ requires e > 0 ∧ \valid(r1+(0..2*e-1)) ∧ \valid(r+(0..2*e-1));
2   @ requires \separated(r+(0..2*e-1),r1+(0..2*e-1));
3   @ requires is_perm(r1,2*e);
4   @ assigns r[0..2*e-1];
5   @ ensures r1[2*e-1] ≠ 2*e-2 ∧
6   @  r1[r1[2*e-1] % 2 == 1 ? r1[2*e-1]-1 : r1[2*e-1]+1] ≠ 2*e-1⇒ is_perm(r,2*e); */
7 int shift(int r1[], int r[], int e);
```

Listing 9.8 – Contrat de la fonction `shift`.

La postcondition lignes 5-6 comporte une reformulation de la condition `(s != d-2 && t != d-1)` sous laquelle cette fonction effectue le décalage de la racine et déclare que, sous cette condition, le tableau r en sortie encode bien une permutation de taille $2e$.

En complément de cette preuve, des tests sur les tableaux générés par les fonctions `isthmic`, `loop` et `shift` sont effectués selon la méthodologie déjà détaillée dans les parties 9.1.3 et 9.2.3. Grâce à la fonction `variant_rpm_atlas` reproduite dans le listing 9.9, nous construisons un atlas de tableaux générés par les fonctions `isthmic`, `loop` et `shift`.

Les cartes locales générées par la fonction `variant_rpm_atlas` sont stockées dans un tableau d'entiers rpm à trois dimensions (supposé pré-alloué en mémoire). Chaque ligne $rpm[e]$ de ce tableau est destinée à stocker les cartes planaires locales à e arêtes (avec $e \leq emax$) produites par les fonctions `isthmic`, `loop` et `shift`.

La fonction `variant_rpm_atlas` diffère de la fonction `rpm_atlas` uniquement dans le cas non isthmique (lignes 23-38). Pour chaque tableau représentant une carte locale planaire ayant $e-1$ arêtes, la fonction `loop` est appelée pour construire une première carte locale planaire à e arêtes (ligne 24). A partir de ce tableau, la fonction `shift` est appelée au sein d'une boucle pour construire successivement d'autres tableaux représentant des cartes locales planaires à e arêtes par décalage de la racine le long de la face extérieure (ligne 31).

Pour chaque tableau généré, nous validons qu'il encode la permutation R d'une carte locale, que cette carte est différente de toutes celles déjà générées, et qu'elle est planaire (lignes 15-17, 25-27 et 32-34).

De plus, nous effectuons pour chaque taille e une comparaison entre le nombre de cartes générées et `count168(e)`, afin de valider que chaque ligne du tableau rpm est complètement remplie (ligne 39).

```
1 void variant_rpm_atlas(int ***rpm, int emax) {
2  int d,e,e1,e2,k;
3  long i,c1,c2,i1,i2;
4
5  for (e = 1; e ≤ emax; e++) {
6   d = 2*e;          // number of darts
7   i = 0;
8   for (e1 = 0; e1 < e; e1++) { // isthmic case
9    e2 = e-1-e1;
10   c1 = count168(e1);
```

```
11      c2 = count168(e2);
12      for (i1 = 0; i1 < c1; i1++) {
13       for (i2 = 0; i2 < c2; i2++) {
14        isthmic(rpm[e1][i1],rpm[e2][i2],rpm[e][i],2*e1,2*e2);
15        if (! b_trans_local(rpm[e][i],d)) exit(1);
16        is_new(rpm,e,i);
17        if (! b_planar_local(rpm[e][i],d)) exit(1);
18        i++;
19       }
20      }
21     }
22     c1 = count168(e-1);
23     for (i1 = 0; i1 < c1; i1++) { // non-isthmic case
24      loop(rpm[e-1][i1],rpm[e][i],e-1);
25      if (! b_trans_local(rpm[e][i],d)) exit(1);
26      is_new(rpm,e,i);
27      if (! b_planar_local(rpm[e][i],d)) exit(1);
28      i++;
29      if (e > 1) {
30       do {
31        k = shift(rpm[e][i-1],rpm[e][i],e);
32        if (! b_trans_local(rpm[e][i],d)) exit(1);
33        is_new(rpm,e,i);
34        if (! b_planar_local(rpm[e][i],d)) exit(1);
35        i++;
36       } while (k ≠ 0);
37      }
38     }
39     if (i ≠ count168(e)) exit(1);
40    }
41   }
```

Listing 9.9 – Fonction `variant_rpm_atlas`.

9.3.4 Décomposition des cartes planaires enracinées

Nous donnons à présent une seconde caractérisation complète d'une carte planaire enracinée selon son nombre d'arêtes, utilisant les opérations c_I, c_L et c_S.

Théorème 11. *Soit M une carte locale planaire ayant $e(M)$ arêtes et de degré de face extérieure (resp. intérieure) $o(M)$ (resp. $i(M)$). M vérifie alors exactement l'un des cas suivants :*

1. *M est la carte vide et $e(M) = o(M) = i(M) = 0$.*
2. *$M = c_I(M_1, M_2)$ pour deux cartes M_1 et M_2 telles que $e(M) = 1 + e(M_1) + e(M_2)$ et $i(M) = o(M) = 2 + o(M_1) + o(M_2)$.*
3. *$o(M) = 1$ et $M = c_L(M_1)$ pour une carte M_1 telle que $e(M) = 1 + e(M_1)$ et $i(M) = o(M_1) + 1$.*
4. *$M = c_S(M_1)$ pour une carte non isthmique M_1 telle que $i(M_1) = i(M) + 1 \geq 2$, $e(M) = e(M_1)$ et $o(M) = o(M_1) + 1$.*

Nos investigations dans la littérature en combinatoire des cartes ne nous ont pas permis d'identifier une référence décrivant cette décomposition. Cependant, la technique selon laquelle nous l'avons dérivée du théorème 10 est classique en combinatoire.

Comme pour les théorèmes 9, et 10, nous validons ce théorème par énumération, en montrant que la fonction `variant_rpm_atlas` construit exactement une carte locale planaire par classe d'isomorphisme enraciné, c'est-à-dire exactement un représentant de chaque carte combinatoire planaire enracinée non étiquetée.

De manière analogue à la validation des théorème 9 et 10, cette validation s'effectue en remplaçant la fonction `is_new_local` par une fonction `is_not_iso_local` dans les lignes 14, 23 et 30 du listing 9.9.

Pour chaque carte planaire locale (d'une taille donnée) produite, cette fonction génère toutes les cartes planaires locales qui lui sont isomorphes par un isomorphisme enraciné préservant l'involution sans point fixe locale L (détaillé dans la partie 9.1.4) et vérifie qu'aucune d'entre elles n'a été stockée précédemment dans le tableau rpm.

Validation	Nombre d'arêtes	Nombre de cartes générées	Nombre de cartes stockées	Durée de calcul (s)
Transitivité	9	19 512 128	19 512 128	9
Planarité	9	19 512 128	19 512 128	14
Unicité	7	235 910	235 910	234
Non isomorphisme	5	117 846 597	3 360	16

TABLE 9.3 – Statistiques de test de la génération de cartes planaires enracinées avec les opérations `isthmic`, `loop` et `shift`.

Des statistiques de test de la génération de cartes planaires enracinées avec les opérations `isthmic`, `loop` et `shift` sont présentées dans la table 9.3. La première colonne indique la propriété validée, la deuxième donne le nombre maximal d'arêtes pour lequel chaque validation a été faite, les troisième et quatrième colonnes donnent respectivement le nombre total de cartes générées et stockées, tandis que la dernière colonne indique la durée de calcul correspondante (en secondes).

9.4 SYNTHÈSE ET PERSPECTIVES

Nous avons présenté dans ce chapitre des implémentations C des opérations de construction de cartes présentées dans le chapitre 8. Nous avons également implémenté des opérations de construction de cartes planaires, dont une variante originale. La preuve formelle automatique qu'elles préservent les permutations a été effectuée, ainsi que des validations supplémentaires par BET, grâce à une génération de cartes jusqu'à une certaine taille. De plus, nous avons énoncé des caractérisations complètes des cartes ordinaires enracinées (planaires ou non), et nous avons proposé des générateurs exhaustifs bornés de ces objets non étiquetés, fondés sur ces caractérisations.

Les opérations de construction de cartes sont toutes basées sur des opérations agissant sur les permutations, implémentées et vérifiées automatiquement dans le chapitre 4. Ce fait est ici primordial. En effet, une implémentation directe en C des opérations sur les cartes, sans utiliser les implémentations des opérations sur les permutations, donne des fonctions dont la preuve est beaucoup plus délicate à effectuer. Dans ce cas, il faut en effet multiplier les invariants, ajouter des assertions ACSL dans le code, et décomposer les postconditions des contrats. Les preuves résultantes demandent alors beaucoup plus de ressources aux prouveurs, sont beaucoup plus longues, et certaines postconditions demeurent non prouvées.

Par rapport aux travaux de Walsh et Lehman [Walsh et al., 1972a] et d'Arquès et Béraud [Arquès et al., 2000], notre travail propose une décomposition des cartes enracinées uniquement selon leur nombre d'arêtes. De plus, nous avons implémenté et testé l'opération inverse de construction de ces cartes.

Ce chapitre n'a pas fait l'objet d'une publication, car nous souhaitons compléter le sujet

avec des preuves formelles des théorèmes de décomposition de cartes. Les parties sur les cartes planaires ont été rédigées en anglais conjointement avec A. Giorgetti et N. Zeilberger. Cette rédaction a été soumise et refusée dans une conférence internationale en septembre 2015.

Une perspective est de démontrer interactivement (par exemple en Coq) ce qu'on valide ici par BET.

10

Hypercartes enracinées et permutations connectées

Dans le chapitre 9, nous avons généré des cartes non étiquetées, planaires ou non, en utilisant la notion de carte locale. Afin de compléter notre collection de générateurs, nous traitons ici le cas des hypercartes enracinées (non étiquetées), selon un autre mode de génération, fondé sur les permutations connectées (voir définition 5 du chapitre 2).

En 2004, P. Ossona de Mendez et P. Rosenstiehl ont exhibé une bijection entre les hypercartes combinatoires enracinées (non étiquetées) possédant d brins et les permutations connectées à $d+1$ éléments [Ossona de Mendez et al., 2004]. En 2009, R. Cori a présenté cette bijection en termes plus simples [Cori, 2009, partie 2.3.1]. Sous le nom d'algorithme OMR (pour Ossona Mendez Rosenstiehl), R. Cori a proposé une version légèrement modifiée de l'algorithme issu de [Ossona de Mendez et al., 2004], permettant de construire une hypercarte enracinée à partir d'une permutation connectée, en utilisant deux opérations.

Nous donnons dans ce chapitre une implémentation en C des opérations composant cette dernière version de l'algorithme. Nous la spécifions formellement, puis nous effectuons des preuves et des tests pour renforcer la confiance dans la correction de cette implémentation.

Ces opérations sont exposées dans la partie 10.1. Notre implémentation en C de ces opérations est présentée dans la partie 10.2. La preuve formelle qu'elles produisent des permutations et des tests supplémentaires sont respectivement détaillés dans les parties 10.3 et 10.4. Un théorème de construction d'hypercartes enracinées est ensuite donné dans la partie 10.5. Des perspectives sont proposées dans la partie 10.6.

Les fichiers sources des générateurs présentés dans ce chapitre sont disponibles dans le répertoire `maps/rhm/` de la bibliothèque `enum`.

10.1 Algorithme OMR

Nous exposons dans cette partie l'algorithme OMR, qui se réduit à l'action de deux opérations sur les permutations. L'une de ces opérations utilise la notion d'indice de maximum partiel d'une permutation, présentée ci-après.

Définition 27 : Indice de maximum partiel

Les indices de maximums partiels (ou left-to-right maxima) d'une permutation $p \in S_n$ sont les indices $i_0, i_1, \ldots, i_k$ $(0 \leq k < n)$ vérifiant $(\forall j.\ j < i_k \Rightarrow p(j) < p(i_k))$ et $i_0 < i_1 < \ldots < i_k$.

En d'autres termes, i est un indice de maximum partiel de p si $p[i]$ est le maximum du préfixe $p[0..i]$ de p. Notons que nous avons nécessairement $i_0 = 0$ et $p(i_k) = n - 1$.

Exemple [23]. *La permutation* $\begin{pmatrix} \mathbf{0} & \mathbf{1} & 2 & 3 & \mathbf{4} & 5 & \mathbf{6} & 7 & 8 & 9 \\ \mathbf{0} & \mathbf{7} & 5 & 4 & \mathbf{8} & 3 & \mathbf{9} & 1 & 2 & 6 \end{pmatrix}$ *a 4 indices de maximums partiels :* $i_0 = \mathbf{0}, i_1 = \mathbf{1}, i_2 = \mathbf{4}$ *et* $i_3 = \mathbf{6}$.

Conformément à nos conventions, cet algorithme est ici présenté avec des permutations connectées de domaine $\{0, \ldots, d - 1\}$, contrairement à [Cori, 2009] où ce domaine est $\{1, \ldots, d\}$. Dans [Cori, 2009], l'algorithme est appliqué à une permutation vue en tant que produit de cycles. Nous adaptons cette présentation au domaine $\{0, \ldots, d - 1\}$ et nous la complétons avec une présentation où la permutation est vue par sa représentation en une ligne, pour bien appréhender son implémentation sous forme de fonction C.

Soit p une permutation de taille $d > 0$, dont les indices de maximums partiels sont i_0, i_1, ..., i_k $(0 \leq k < d)$. Nous définissons deux opérations c_L et c_R agissant sur p, définies ci-après.

L'opération c_L construit une permutation de taille $d - 1$ en supprimant la valeur maximale $d - 1$ de son cycle de p.

L'opération c_R construit la permutation de taille $d - 1$ égale au produit de cycles

$$(0\ 1\ \cdots\ i_1 - 1)\ (i_1\ i_1 + 1\ \cdots\ i_2 - 1)\ \ldots\ (i_k\ i_k + 1\ \cdots\ d - 2)$$

dont la notation sur deux lignes est

$$\begin{pmatrix} 0 & 1 & \cdots & \cdots & i_1 - 1 & i_1 & \cdots & \cdots & i_2 - 1 & \cdots & i_k & \cdots & \cdots & d - 2 \\ 1 & 2 & \cdots & i_1 - 1 & 0 & i_1 + 1 & \cdots & i_2 - 1 & i_1 & \cdots & i_k + 1 & \cdots & d - 2 & i_k \end{pmatrix}.$$

Dans la suite, on utilisera surtout la notation de $c_R(p)$ sur une ligne, qui est

$$1\quad 2\quad \cdots\quad i_1 - 1\quad 0\quad i_1 + 1\quad \cdots\quad i_2 - 1\quad i_1\quad \cdots\quad i_k + 1\quad \cdots\quad d - 2\quad i_k.$$

L'algorithme OMR consiste à appliquer les opérations c_L et c_R sur une permutation connectée de taille $d > 0$ pour construire deux permutations R et L de taille $d - 1$, de telle sorte que le couple de permutations (R, L) obtenu agit transitivement sur $D = \{0, \ldots, d-2\}$.

Exemple [24]. *A partir de la permutation connectée*

$$p = (0)\ (1\ 7)\ (2\ 5\ 3\ 4\ 8)\ (6\ 9) = 0\ 7\ 5\ 4\ 8\ 3\ 9\ 1\ 2\ 6 \in S_{10},$$

l'action de c_L *sur* p *permet de construire la permutation*

$$L = (0)\ (1\ 7)\ (2\ 5\ 3\ 4\ 8)\ (6) = 0\ 7\ 5\ 4\ 8\ 3\ 6\ 1\ 2 \in S_9$$

en supprimant le maximum 9 *de son cycle* $(6\ 9)$ *de* p.

La permutation p *possède 4 indices de maximums partiels* $i_0 = \mathbf{0}, i_1 = \mathbf{1}, i_2 = \mathbf{4}$ *et* $i_3 = \mathbf{6}$. *L'action de* c_R *sur* p *permet de construire la permutation*

$$R = (\mathbf{0})\ (\mathbf{1}\ 2\ 3)\ (\mathbf{4}\ 5)\ (\mathbf{6}\ 7\ 8) = 0\ 2\ 3\ 1\ 5\ 4\ 7\ 8\ 6 \in S_9$$

qui s'obtient comme produit de 4 cycles, chaque cycle commençant par un indice de maximum partiel et contenant les nombres suivants, jusqu'à l'indice de maximum partiel suivant exclu.

10.2 Implémentation en C

La permutation L est obtenue à partir d'une permutation connectée p de taille d par l'action de l'opération c_L, en supprimant $d-1$ de son cycle de p. L'opération de suppression du maximum d'une permutation, qui effectue cette tâche, a été présentée dans le chapitre 4. Elle est implémentée en C par la fonction `remove_max`, et le fait qu'elle préserve les permutations a été prouvé automatiquement.

Il existe différentes manières d'implémenter l'opération c_R permettant de construire la permutation R à partir d'une permutation connectée p de taille d. Nous réutilisons ici l'opération sum de somme directe de permutations définie dans le chapitre 4, dont l'implémentation en C a été formellement vérifiée. Soit $i_0, i_1, \ldots, i_k$ $(0 \leq k \leq d-1)$ les $k+1$ indices de maximums partiels de p. La permutation R associée possède $k+1$ cycles. Elle peut être obtenue à partir de la permutation circulaire $(0\ 1\ \cdots\ i_1 - 1)$, par k sommes directes successives avec les permutations circulaires $(0\ 1\ \cdots\ i_2 - i_1 - 1)$, ... et $(0\ 1\ \cdots\ d - i_k - 2)$. Ces permutations circulaires de la forme $(0\ 1\ \cdots\ n-1)$ sont construites par la fonction `cycle`, présentée dans le listing 10.1 lignes 1-4. Cette fonction construit la permutation circulaire $p = (0\ 1\ \cdots\ n-1)$ de taille $n > 0$, dont les éléments sont les n premiers entiers, telle que $p(i) = i + 1$ pour tout entier $i \in \{0, \ldots, n-2\}$ et $p(n-1) = 0$.

```
1  void cycle(int p[], int n) {
2   for(int i = 0; i < n-1; i++) p[i] = i+1;
3   p[n-1] = 0;
4  }
5
6  void p2R(int p[], int r[], int d) {
7   int j = 0, m = p[0];
8   for (int i = 1; i < d; i++) {
9    if (p[i] > m) {
10    cycle(p,i-j); sum_inplace(r,p,j,i-j);
11    j = i;
12    m = p[j];
13   }
14  }
15  if (j < d-1) { cycle(p,d-j-1); sum_inplace(r,p,j,d-j-1); }
16 }
```

Listing 10.1 – Fonction `p2R`.

La fonction `p2R`, présentée également dans le listing 10.1 lignes 6-16, “assemble” ensuite les permutations circulaires construites par la fonction `cycle`. Cette fonction prend en entrée une permutation p de taille $d > 0$, et construit la permutation R des sommets de l'hypercarte correspondante , représentée par le tableau r. Lors d'un parcours du tableau p dans une boucle d'indice i (ligne 8), chaque indice de maximum partiel est stocké dans une variable j (initialisée à 0), et chaque maximum correspondant est stocké dans une variable m (initialisée à $p[0]$). A chaque nouvel indice de maximum partiel détecté, la fonction `p2R` construit dans le tableau p une permutation circulaire de taille $i - j$ en appelant la fonction `cycle`, puis effectue la somme directe de la permutation R courante de taille j avec le préfixe $p[0..i-j-1]$ résultant, en appelant la fonction `sum_inplace` stockant cette somme dans R (ligne 10). Le préfixe $p[0..i-j-1]$, qui n'est plus utilisé par la suite, est donc écrasé à chaque appel de la fonction `cycle`. Après l'exécution de cette

boucle, la permutation R est de taille j et le dernier cycle de R à construire est de taille $n-j-1$. Si $j < d-1$, la construction de R se termine par un nouvel appel aux fonctions `cycle` puis `sum_inplace` (ligne 15). La permutation R obtenue est de taille $d-1$.

Exemple [25]. *Considérons à nouveau la permutation connectée*

$$p = (0)\ (1\ 7)\ (2\ 5\ 3\ 4\ 8)\ (6\ 9) = 0\ 7\ 5\ 4\ 8\ 3\ 9\ 1\ 2\ 6 \in S_{10}$$

qui a pour indices de maximums partiels $i_0 = 0, i_1 = 1, i_2 = 4$ et $i_3 = 6$. Le déroulement de l'exécution de la fonction `p2R` appelée avec cette permutation p est donné dans la table 10.1.

i	j	m	p en ligne	R en ligne	R en cycles
1	0	0	**0** 7 5 4 8 3 9 1 2 6	**0**	(**0**)
4	1	7	**1 2 0** 4 8 3 9 1 2 6	0 **2 3 1**	(0) (**1 2 3**)
6	4	8	**1 0** 0 4 8 3 9 1 2 6	0 2 3 1 **5 4**	(0) (1 2 3) (**4 5**)
10	6	9	**1 2 0** 4 8 3 9 1 2 6	0 2 3 1 5 4 **7 8 6**	(0) (1 2 3) (4 5) (**6 7 8**)

TABLE 10.1 – Exemple de déroulement de l'exécution de la fonction `p2R`.

Les trois premières colonnes indiquent les valeurs des variables i, j et m au début des itérations de la boucle `for` qui satisfont la condition du `if` de la ligne 9, sauf dans la dernière ligne de la table, qui indique leurs valeurs en sortie de boucle, pour la construction du dernier cycle de R ligne 15. Les valeurs en gras indiquent les cycles construits dans p à chaque appel de la fonction `cycle` et formant un nouveau cycle de R à chaque appel de la fonction `sum_inplace`.

10.3 SPÉCIFICATION ET PREUVES

Si le tableau p en entrée des fonctions `cycle` et `p2R` est une permutation de taille d, avec $d > 0$, alors leur sortie est une permutation. La condition que p soit connectée est inutile pour démontrer cette propriété, que nous appelons préservation des permutations. Cette condition n'est utile que pour établir que le couple (R,L) de permutations produit par l'algorithme OMR agit transitivement sur son support $\{0,\ldots,d-2\}$. Nous spécifions et démontrons ici formellement et automatiquement cette propriété de préservation des permutations.

Le listing 10.2 montre les fonctions `cycle` et `p2R` spécifiées en ACSL. La spécification et la preuve de la préservation des permutations par la fonction `p2R` s'effectuent en deux temps.

Nous spécifions tout d'abord la fonction `cycle` de manière similaire à d'autres fonctions présentées dans ce mémoire. La postcondition ligne 3 (déclarant que le tableau p construit est une permutation de taille d) est établie grâce aux invariants de boucle des lignes 7 et 8.

Le contrat de la fonction `p2R` stipule que si p est une permutation de taille $d > 0$, alors le tableau q construit par cette fonction encode une permutation de taille $d-1$. La preuve de la fonction `p2R` peut alors s'effectuer grâce aux contrats formellement prouvés des fonctions `cycle` et `sum_inplace`.

```
1  /*@ requires n > 0 ∧ \valid(p+(0..n-1));
2    @ assigns p[0..n-1];
3    @ ensures is_perm(p,n); */
4  void cycle(int p[], int n) {
5
6   /*@ loop invariant 0 ≤ i ≤ n-1;
7     @ loop invariant is_cycle(p,i);
8     @ loop invariant is_fct(p,i,n) ∧ is_linear(p,i);
9     @ loop assigns i, p[0..n-2];
10    @ loop variant n-1-i; */
11  for(int i = 0; i < n-1; i++) p[i] = i+1;
12  p[n-1] = 0;
13 }
14
15 /*@ requires d > 0 ∧ \valid(q+(0..d-1)) ∧ \valid(p+(0..d-1));
16   @ requires \separated(p+(0..d-1),q+(0..d-1)) ∧ is_perm(p,d);
17   @ assigns p[0..d-1],q[0..d-1];
18   @ ensures is_perm(q,d-1); */
19 void p2R(int p[], int q[], int d) {
20  int i, j = 0, m = p[0];
21
22  /*@ loop invariant 1 ≤ i ≤ d∧ 0 ≤ j < d∧ 0 < i-j ≤ d;
23    @ loop invariant is_perm(q,j);
24    @ loop assigns i,j,m,p[0..d-1],q[0..d-1];
25    @ loop variant d-i; */
26  for (i = 1; i < d; i++) {
27   if (p[i] > m) {
28    cycle(p,i-j); sum_inplace(q,p,j,i-j);
29    j = i;
30    m = p[j];
31   }
32  }
33  if (j < d-1) { cycle(p,d-j-1); sum_inplace(q,p,j,d-j-1); }
34 }
```

Listing 10.2 – Fonctions `cycle` et `p2R` annotées.

En effet, nous disposons d'une preuve formelle que les fonctions `cycle` et `sum_inplace` préservent les permutations. Il convient donc de s'assurer que cette préservation s'effectue à chaque itération de la boucle de la fonction `p2R`, qui comporte des appels aux fonction `cycle` et `sum_inplace`. Les contrats des fonctions `cycle` et `sum_inplace` permettent de (i) satisfaire la précondition

```
requires is_perm(p,d);
```

de la fonction `sum_inplace` appelée dans cette boucle, et (ii) prouver la préservation de l'invariant de boucle

```
loop invariant is_perm(q,j);
```

ligne 23. Cet invariant permet alors de prouver la postcondition

```
ensures is_perm(q,d-1);
```

du contrat ACSL de la fonction `p2R`, ce qui permet d'établir que chaque tableau p construit par cette fonction encode une permutation.

L'invariant

```
loop invariant 1 ≤ i ≤ d∧ 0 ≤ j < d∧ 0 < i-j ≤ d;
```

est nécessaire, car il permet de s'assurer que la variable de boucle i, la variable j et la taille $i - j$ de la permutation circulaire p (construite par la fonction `cycle`) sont correctement bornées.

L'implémentation directe en C de l'opération c_R, sans utiliser l'implémentation (prouvée automatiquement) de l'opération *sum* sur les permutations, est beaucoup plus difficile à spécifier, et certaines postconditions demeurent non prouvées automatiquement. A nouveau, comme dans le chapitre 9, la modularité des preuves est ici essentielle, par l'usage de fonctions du chapitre 4 formellement vérifiées. Le prix à payer pour obtenir une fonction dont la correction est prouvée est quasi-nul ici, puisque la perte d'efficacité due à l'usage des fonctions `cycle` et `sum_inplace`, par rapport à la version “directe”, est minime. A titre d'exemple, les hypercartes possédant 11 brins sont générées à partir des permutations connectées de taille 12 en 2 minutes et 50 secondes avec le code présenté ici, et en 2 minutes et 20 secondes avec la version “directe”.

Lors de la preuve formelle de la fonction `p2R`, les solveurs Alt-Ergo, CVC3 et CVC4 déchargent 50 obligations de preuves en 13 secondes.

10.4 TESTS

En complément, des validations sur les tableaux générés par les fonctions `p2R` et `remove_max` ont été effectuées par BET. Dans ce but, nous avons conçu un générateur séquentiel d'hypercartes enracinées composé des fonctions `first_rhm` et `next_rhm`, présentées dans le listing 10.3.

```
1 int first_rhm(int p[], int r[], int l[], int d) {
2  first_cperm(p,d+1);
3  copy(p,p1,d+1);
4  p2R(p1,r,d+1);
5  remove_max(p,d+1,l);
6  return 1;
7 }
8
9 int next_rhm(int p[], int r[], int l[], int d) {
10  int tmp;
11
12  tmp = next_cperm(p,d+1);
13  if (tmp == 0) return 0;
14  copy(p,p1,d+1);
15  p2R(p1,r,d+1);
16  remove_max(p,d+1,l);
17  return 1;
18 }
```

Listing 10.3 – Générateur séquentiel d'hypercartes.

Ce générateur construit chaque permutation connectée de taille $d+1>0$ grâce au générateur séquentiel formellement vérifié CPERM de permutations connectées, présenté dans le chapitre 3. Un appel à la fonction `p2R` construit ensuite la permutation R (encodée par le tableau r) de taille d associée, puis un appel à la fonction `remove_max` construit la permutation L (encodée par le tableau l) de taille d associée. La fonction `p2R` modifiant le tableau p, il est nécessaire d'effectuer une copie de p dans un tableau global p_1 qui est passé en paramètre de la fonction `p2R`, pour que p reste intact.

```
1 void rhm_test(int **rhm_r, int **rhm_l, int d) {
2  long i;
3  int p[d];
4
5  i = 0;
6  first_rhm(p,rhm_r[i],rhm_l[i],d);
7  if (! b_trans(rhm_r[i],rhm_l[i],d)) exit(1);
8  is_new_rhm(rhm_r,rhm_l,d,i);
```

```
 9  i++;
10  while(next_rhm(p,rhm_r[i],rhm_l[i],d) ≠ 0) {
11   if (! b_trans(rhm_r[i],rhm_l[i],d)) exit(1);
12   is_new_rhm(rhm_r,rhm_l,d,i);
13   i++;
14  }
15 }
```

Listing 10.4 – Fonction `rhm_test`.

Il reste à valider que les couples (R, L) de permutations de taille d ainsi produits sont transitifs, et qu'ils sont deux à deux non équivalents par un isomorphisme enraciné. Dans ce but, nous implémentons la fonction `rhm_test` reproduite dans le listing 10.4, qui construit et stocke tous des couples de tableaux (d'une taille donnée) générés par les fonctions `first_rhm` et `next_rhm`.

A la différence des générations de cartes effectuées dans le chapitre 9, la fonction `rhm_test` génère séquentiellement des couples de tableaux d'une taille donnée, sans générer de tableaux de tailles inférieures. Les couples de tableaux de longueur d générés par la fonction `rhm_test` sont stockés dans deux tableaux d'entiers rhm_r (pour les permutations R des sommets) et rhm_l (pour les permutations L des brins) à deux dimensions (supposés pré-alloués en mémoire). L'élément $rhm_r[i]$ (resp. $rhm_l[i]$) stocke la composante R (resp. L) de la i-ème hypercarte à d brins produite par les fonctions `first_rhm` et `next_rhm`. La taille de la première dimension du tableau est fixée au nombre d'hypercartes enracinées à d brins égal à $A003319(d+1)$. Ce choix sera justifié dans la partie 10.5.

La fonction booléenne `b_trans` (présentée dans le chapitre 5) permet alors de tester que le couple (R, L) obtenu est transitif et forme donc bien une hypercarte (lignes 7 et 11). Nous validons également que chaque couple de permutations généré est différent de tous ceux générés précédemment, par comparaison de leurs valeurs, grâce à la fonction `is_new_rhm` (lignes 8 et 12).

10.5 Caractérisation des hypercartes enracinées

Nous énonçons dans le théorème 12 une caractérisation complète des hypercartes enracinées.

Théorème 12 ([Ossona de Mendez et al., 2004])**.** *La famille des hypercartes enracinées à $d \geq 0$ brins est en bijection avec la famille des permutations connectées de taille $d+1$, par la fonction qui associe l'hypercarte étiquetée $(\{0, \ldots, d-1\}, c_R(p), c_L(p))$ à la permutation connectée p de taille $d+1$.*

Une preuve formelle de ce théorème est laissée en perspective. Auparavant, nous le validons ici en complétant les preuves et tests des parties 10.3 et 10.4 par un BET de la propriété de non-isomorphisme (enraciné) selon laquelle la fonction `rhm_test` construit exactement une hypercarte étiquetée par classe d'isomorphisme enraciné, c'est-à-dire exactement un représentant de chaque hypercarte combinatoire enracinée non étiquetée.

Cette validation s'effectue en remplaçant la fonction `is_new_rhm` par une fonction `is_not_iso_rhm` dans les lignes 8 et 12 du listing 10.4. Pour chaque hypercarte (d'une taille donnée) produite, cette fonction génère toutes les hypercartes qui lui sont isomorphes par

un isomorphisme enraciné, et vérifie qu'aucune d'entre elles n'a été stockée précédemment dans les tableaux rhm_r et rhm_l.

A cette fin, nous utilisons un générateur séquentiel de permutations sur $\{0, \ldots, d-1\}$ préservant la racine $(d-1)$. Ce générateur produit $(d-1)!$ permutations θ de taille d telles que $\theta(d-1) = d-1$. En conjuguant chaque composante de l'hypercarte générée par la fonction `rhm_test` avec chaque renommage autre que l'identité produit par ce générateur, la fonction `is_not_iso_rhm` génère les $(d-1)! - 1$ autres hypercartes étiquetées qui lui sont isomorphes, puis valide leur absence dans les tableaux rhm_r et rhm_l à un indice inférieur.

En complément, nous comptons le nombre d'hypercartes de taille d générées, et nous le comparons au $(d+1)$-ième terme de la séquence A003319 [The OEIS Foundation Inc., 2010], afin d'établir que toutes les hypercartes enracinées de taille d ont bien été générées.

Validation	Nombre de brins	Nombre d'hypercartes générées	Nombre d'hypercartes stockées	Durée de calcul (s)
Transitivité	8	273 343	273 343	1
Unicité	8	273 343	273 343	336
Non isomorphisme	7	149 167 669	29 093	1 680

TABLE 10.2 – Statistiques de validation de la génération séquentielle d'hypercartes enracinées.

Des statistiques de validation de la génération séquentielle d'hypercartes enracinées sont présentées dans la table 10.2. La première colonne indique la propriété validée, la deuxième donne le nombre de brins de chaque validation, les troisième et quatrième colonnes donnent respectivement le nombre total d'hypercartes générées et stockées, tandis que la dernière colonne donne la durée de calcul correspondante (en secondes).

Ces validations nous permettent d'affirmer que la fonction `rhm_test` est un générateur exhaustif borné d'hypercartes enracinées (non étiquetées). D'autre part, associées au comptage des hypercartes générées, ces validations permettent également de valider la bijection énoncée dans le théorème 12.

10.6 PERSPECTIVES

Nous avons présenté dans ce chapitre une approche de construction d'hypercartes par deux opérations appliquées aux permutations connectées. Nous avons implémenté ces opérations de construction d'hypercartes et prouvé automatiquement qu'elles préservent les permutations. Des validations supplémentaires ont été effectuées par BET permettant de proposer un générateur testé d'hypercartes enracinées (non étiquetées).

Il reste beaucoup à faire concernant cette approche. R. Cori évoque dans [Cori, 2009, Section 5.2.] une autre bijection entre les cartes enracinées (non étiquetées) et les involutions sans point fixe connectées. Cette correspondance, proche de celle entre permutations connectées et hypercartes enracinées non étiquetées, utilise deux opérations similaires à celles présentées dans ce chapitre. Énumérer les cartes enracinées par le biais de ces opérations est une perspective que nous n'avons pas pu aborder dans le cadre de ce travail faute de temps. Une autre perspective est d'implémenter des opérations de construction de cartes via une bijection entre cartes enracinées et

mots à double occurrence connectés, décrite par P. Ossona De Mendez et P. Rosenstiehl dans [Ossona De Mendez et al., 2006]. L'implémentation de l'inverse de l'algorithme OMR, décrit dans [Cori, 2009], et permettant d'associer une permutation connectée de taille $d + 1$ à toute hypercarte non étiquetée à d brins, constitue une autre perspective intéressante.

11

CONCLUSION

Pour conclure ce travail, nous proposons dans la partie 11.1 une synthèse permettant d'apporter des éléments de réponse à la problématique de cette étude. Quelques perspectives relatives à nos différentes contributions sont ensuite données dans la partie 11.2.

11.1 SYNTHÈSE

Ce travail de thèse a été avant tout un champ d'expérimentations diverses. L'approche résolument expérimentale adoptée dans ce travail a débouché sur la conception de la bibliothèque `enum`, disponible sous forme d'une archive `enum.*.tar.gz` téléchargeable depuis la page http://members.femto-st.fr/richard-genestier/en. . Cette bibliothèque constitue un outil facilement utilisable par tout programmeur, et peut répondre à plusieurs attentes. Elle contient des générateurs formellement vérifiés, utiles pour tester d'autres algorithmes que ceux proposés dans la bibliothèque ; ils jouent dans ce cas le rôle d'outils certifiés de test. De plus, la bibliothèque `enum` contient des patrons de génération ainsi que des générateurs génériques. Ainsi, elle permet également de concevoir facilement des générateurs de nouvelles familles combinatoires, par instanciation des patrons ou des générateurs génériques fournis.

Nos travaux montrent comment les méthodes et outils de preuve de programmes peuvent être intégrés à la conception de programmes de génération de données structurées, avec un coût limité. De plus, une analyse de notre catalogue de preuves automatiques permet de mieux appréhender les limites des solveurs SMT actuels. Enfin, les formalisations des cartes que nous avons adoptées nous ont permis d'obtenir une collection de générateurs efficaces certifiés de cartes (planaires ou non) et d'hypercartes non étiquetées. Ainsi, nous pouvons considérer que les trois objectifs de cette thèse, présentés dans la problématique générale de l'introduction, sont atteints.

Globalement, notre travail montre la fertilisation croisée de la combinatoire énumérative et des méthodes du génie logiciel. Par exemple, les décompositions de cartes présentées dans le chapitre 9 sont issues des efforts de formalisation et de preuve formelle effectués sur des décompositions connues des cartes. De plus, l'étude combinatoire théorique sur les mots a été grandement facilitée par la mise au point d'un générateur de mots conçu sur le modèle des générateurs exhaustifs bornés de tableaux structurés. La combinatoire énumérative, quant à elle, constitue une source inépuisable de problématiques qui permettent de faire évoluer les outils du génie logiciel. Par exemple, les différentes op-

tions de formalisation des mots de Dyck, évoquées dans la partie 6.3 du chapitre 6, ont soulevé le problème du raisonnement sur les sommes généralisées dans un tableau d'entiers. Une réponse à ce problème pourrait venir de l'article [Daca et al., 2016] paru à la fin de la thèse. En effet, cet article présente une nouvelle procédure de décision permettant d'exprimer des problèmes de comptage dans des tableaux d'entiers. Nous n'avons malheureusement pas eu le temps de déterminer si cette procédure s'applique à la formalisation des (mélanges de) mots de Dyck.

Concernant les formalisations des cartes que nous avons adoptées, il est important de remarquer qu'elles permettent de générer efficacement les cartes enracinées (non étiquetées), ce qui, à notre connaissance, ne semble pas être le cas des autres formalisations existantes.

Nous pouvons à présent répondre aux interrogations énoncées dans l'introduction :

(Q_1) Face à quels types de problèmes peut-on espérer mener à bien une preuve formelle automatique utilisant des solveurs SMT ?

Il résulte de nos expérimentations un certain nombre de constats. Afin de faciliter la démonstration automatique de la correction de ces fonctions C, ces dernières doivent utiliser un petit fragment du langage C, et leurs spécifications doivent pouvoir être exprimées en logique du premier ordre. Plus précisément, il convient de considérer au sein de ces fonctions des tableaux d'entiers alloués préalablement (pas de pointeur ni d'allocation dynamique), et uniquement des expressions d'arithmétique linéaire sur les entiers. Ensuite, les difficultés dans l'établissement des preuves sont liées à la correction et à la précision des invariants de boucles, qui ne doivent être ni trop faibles (ils ne permettent pas de prouver les postconditions attendues), ni trop forts (les prouveurs peinent alors à prouver leur préservation). D'autre part, il convient de porter une attention particulière à la clause `assigns` des invariants de boucle. Dans le cas où il n'est pas possible de spécifier précisément les variables modifiées par les itérations d'une boucle, des invariants supplémentaires sont parfois nécessaires pour faciliter la tâche des prouveurs. Enfin, la présence de disjonctions logiques dans les prédicats utilisés dans les spécifications posent des problèmes de vérification aux solveurs SMT, et doivent donc être si possible limitées.

(Q_2) Quels compromis, quelles adaptations doit-on faire sur le code pour parvenir à prouver automatiquement sa correction ?

De manière générale, lorsque la preuve de la correction d'une fonction présente une difficulté, utiliser la modularité aide grandement les prouveurs dans leur tâche. En effet, le fait de décomposer une fonction en sous-fonctions munies de contrats contenant des postconditions intermédiaires permet de considérer les problèmes individuellement et d'identifier l'erreur ou le défaut de spécification. L'intérêt de la modularité est exposé à plusieurs reprises dans ce mémoire, notamment avec les fonctions sur les permutations présentées dans le chapitre 4 et réutilisées dans différentes circonstances. Néanmoins, ce besoin de modularité peut avoir un (léger) impact sur l'efficacité des fonctions, comme exposé dans le chapitre 10.

(Q_3) Dans le cas où le coût de la preuve devient trop important, quels types de tests effectuer ?

Conformément à l'approche empirique adoptée, nous avons confronté les solveurs SMT à différents problèmes, plus ou moins complexes, afin de tester leurs capaci-

tés réelles. Nous avons notamment approché ces limites lors de la preuve du générateur séquentiel efficace d'involutions sans point fixe, présentée dans le chapitre 5. Les conditions de vérification générées par WP ont nécessité une extension de la durée de calcul des solveurs SMT à 10 minutes, ce qui n'est pas usuel dans ce domaine. D'autre part, l'effort de preuve est ici important et devient déraisonnable lorsqu'une validation par comptage via une génération exhaustive des involutions s'effectue pour une taille élevée ($n = 24$) dans un temps raisonnable (environ 1 heure). On touche ici au problème de la preuve perçue comme frein au développement. Dans cet ordre d'idée, nous n'avons pas spécifié le générateur séquentiel de mélanges de mots de Dyck (présenté dans le chapitre 6), bien que nous disposions d'une formalisation "papier" du comportement de ce générateur. En effet, cette formalisation est plus complexe que celle du générateur de mots de Dyck (présentée également dans le chapitre 6), qui est elle-même assez conséquente, et l'adapter aux mélanges aurait nécessité un temps et un effort déraisonnables. A nouveau, des validations par test exhaustif borné peuvent s'effectuer dans ce cas pour une taille élevée (mélanges de 20 lettres) en quelques minutes.

(Q_4) Quels types de problèmes nécessitent une preuve interactive utilisant des assistants de preuve ?

De manière générale, les spécifications comportant des quantifications existentielles posent des problèmes d'instanciation aux solveurs SMT. On peut parfois contourner ce problème grâce à des assertions indiquant la bonne manière d'instancier ces quantifications, comme décrit dans la méthodologie de preuve de la propriété de progression détaillée dans le chapitre 3. Un autre problème est posé par la présence de quantifications sur des tableaux dans les spécifications (rencontré par exemple lors de la spécification de la fonction `b_trans` caractérisant la transitivité de deux tableaux d'entiers). En effet, les conditions de vérification générées correspondantes sont possiblement non déchargées par les prouveurs automatiques actuels. S'il existe, un fragment décidable de la théorie des tableaux incluant ces conditions de vérification n'est pas encore identifié. Ces situations encouragent l'utilisation d'un assistant de preuve pour les aborder (comme détaillé dans le chapitre 8).

11.2 PERSPECTIVES

Les perspectives de ce travail sont nombreuses, dans chaque thématique abordée, mais aussi pour les relier entre elles.

Tout d'abord, du point de vue logiciel, les démarches d'utilisation de la bibliothèque `enum` sont simples, mais nécessitent dans certains cas de créer de nouveaux fichiers et d'appeler les générateurs ou de les instancier "à la main". Une automatisation de son utilisation est possible, afin de la rendre plus conviviale. Cette évolution de la bibliothèque `enum` constitue une perspective à court terme.

Cette bibliothèque pourrait également évoluer de façon plus radicale. En effet, pour faire face aux limites rencontrées lors de la vérification automatique de certains programmes, une perspective globale serait la transformation de ces programmes impératifs en programmes fonctionnels dans le but d'utiliser l'outil Why3 [Filliâtre et al., 2013], afin d'y effectuer certaines preuves de manière interactive.

Concernant la thématique des cartes, plusieurs perspectives apparaissent. Nous avons formalisé et généré les cartes combinatoires par différentes approches. Nous nous sommes efforcés de rester au niveau de la définition mathématique des cartes, sous forme de permutations. Cependant, dans le cas des cartes planaires non étiquetées, traité dans le chapitre 6, nous avons considéré ces objets via leur codage par des mots, les NCS. Une perspective serait de combler le fossé entre ces deux approches en spécifiant et implémentant une fonction de codage/décodage d'un NCS en une permutation R de la rotation de brins autour des sommets d'une carte locale planaire.

D'autre part, dans la formalisation utilisée pour démontrer le théorème des quatre couleurs en Coq [Gonthier, 2008, Gonthier, 2005], les hypercartes combinatoires sont définies comme triplets d'endofonctions dont le produit est l'identité [Gonthier, 2005, p.19]. Il serait intéressant d'adapter cette idée aux cartes (locales ou non) et de déterminer dans quelle mesure cela pourrait simplifier notre formalisation.

Certains des travaux présentés dans ce mémoire n'ont pas été publiés, comme par exemple les décompositions de cartes, car seule une partie de la preuve formelle de ces décompositions a été fournie. De telles preuves formelles complètes constituent des perspectives à moyen terme.

Enfin, durant cette étude, nous nous sommes concentrés sur les cartes enracinées, mais les cartes non enracinées jouent également un rôle important dans certains domaines de la physique théorique [Planat et al., 2015]. Une perspective plus lointaine serait de générer ce genre d'objets, et d'effectuer des preuves formelles sur certaines de leurs propriétés.

Globalement, ce travail préconise la formalisation machine des problèmes dès le début de leur étude, ainsi d'une utilisation systématique d'outils logiciels et de méthodes de validation. Gageons qu'à l'avenir les méthodes qui assistent – voire automatisent – la validation – voire la découverte – de résultats simples pourront aussi s'appliquer à des problèmes de complexité trop élevée pour être traités sans de tels outils.

Table des figures

2.1 Surfaces de genre 0, 1 et 2. 14
2.2 Différentes représentations d'une carte planaire. 15
2.3 Exemple de carte topologique étiquetée. 16
2.4 Six cartes combinatoires étiquetées appartenant à la même classe d'équivalence. 20
2.5 Les 10 ROMs de taille 2. 21

5.1 Durées de génération de FFIs en fonction de la demi-cardinalité de leur support. 73
5.2 Exemple de carte locale à 6 arêtes. 79
5.3 Durées de génération d'(hyper)cartes en fonction du nombre de brins. 81

6.1 Une carte planaire enracinée avec 7 arêtes et sa représentation sous forme de NCS. 87
6.2 Mots de Dyck de taille 3 avec leur chemin de Dyck. 89
6.3 Exemple de mot de Dyck de taille 6, chemin w et tableau de hauteurs h associés. 90
6.4 Révision du suffixe du tableau de hauteurs présenté dans la figure 6.3, avec son chemin de Dyck associé. 91
6.5 Représentation du mélange de mots de Dyck $ab\bar{a}baa\bar{a}\bar{a}b\bar{b}\bar{b}\bar{b}$ par un chemin composé d'étapes N, S, E et O. 95
6.6 Mélange de mots de Dyck avec son chemin bicolore et ses tableaux de hauteurs. 97
6.7 Exemple de mélange de mots de Dyck (a) et son successeur immédiat (b) après la révision de son suffixe, avec leurs chemins et leurs tableaux de hauteurs. 98
6.8 Exemple de NCS et son successeur après révision de son suffixe, avec leurs chemins et leurs tableaux de hauteurs. 100

8.1 Exemple de construction d'une carte isthmique. 129
8.2 Exemples de construction de cartes non isthmiques. 130

9.1 Illustration de l'opération c_{NP}^k et de la fonction `non_isthmic_rpm`. 145
9.2 Exemple de décalage de la racine par l'opération c_S. 150

9.3 Illustration des cas où la fonction `shift` retourne 0. 151

Liste des tables

3.1 Résultats de vérification des générateurs de la bibliothèque `enum`. 45

3.2 Résultats de vérification pour les algorithmes efficaces avec une assertion finale. 47

4.1 Résultats de vérification des opérations sur les permutations. 64

5.1 Trace d'exécution de la fonction `next_effi` sur les FFI 2 3 0 1 7 8 9 4 5 6 et 1 0 3 2 7 8 9 4 5 6. 67

5.2 Trace d'exécution de la fonction `b_trans` sur les permutations p_1 = (0 4 7 2) (1 5 3) (6) et p_2 = (0 7) (1 5) (2 4 3 6). 75

5.3 Trace d'exécution de la fonction `b_trans` sur les permutations p_1 = (0 4 7 2) (1 5 3) (6) et p_2 = (0) (3 1 6 5) (2 4) (7). 76

5.4 Nombres de cartes de chaque type générées. 80

7.1 Visualisation de certains motifs d'un mélange sur une carte planaire enracinée. 104

7.2 Enumération des classes de π-equivalence pour les mélanges et les cartes planaires enracinées. 104

7.3 Durées de calcul du nombre de classes de π-équivalence des mélanges pour les tailles de 1 à 8. 106

9.1 Statistiques de test de la génération de cartes enracinées. 144

9.2 Statistiques de test de la génération de cartes planaires enracinées avec les opérations `isthmic` et `non_isthmic_rpm`. 149

9.3 Statistiques de test de la génération de cartes planaires enracinées avec les opérations `isthmic`, `loop` et `shift`. 154

10.1 Exemple de déroulement de l'exécution de la fonction `p2R`. 160

10.2 Statistiques de validation de la génération séquentielle d'hypercartes enracinées. 164

Liste des définitions

1 Définition : Permutation 11
2 Définition : Composition de deux permutations 12
3 Définition : Cycle [Comtet, 1970] 12
4 Définition : Conjugaison 12
5 Définition : Permutation connectée [Cori et al., 2012] 13
6 Définition : Carte topologique [Lando et al., 2004] 14
7 Définition : Action transitive 17
8 Définition : Carte combinatoire ordinaire étiquetée 17
9 Définition : isomorphisme enraciné 19
10 Définition : Carte combinatoire enracinée 19
11 Définition : Carte combinatoire générale étiquetée 21
12 Définition : Hypercarte combinatoire étiquetée 22

13 Définition : Ordre lexicographique 34
14 Définition : Fonction à croissance limitée 35

15 Définition : Insertion dans une permutation 54
16 Définition : Retrait du maximum dans une permutation 55
17 Définition : Somme directe de deux permutations 55

18 Définition : Involution sans point fixe locale 66
19 Définition : Carte locale 79

20 Définition : Mot de Dyck 84
21 Définition : Longueur et taille d'un mot de Dyck 84
22 Définition : Mélange de deux mots de Dyck 85
23 Définition : Mélange non croisé 86
24 Définition : Tableau de hauteurs 89
25 Définition : Tableaux de hauteurs d'un mélange 96

26 Définition : Relation de π-équivalence 103

27 Définition : Indice de maximum partiel 158

Bibliographie

[Ahrendt et al., 2014] Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P. H., et Ulbrich, M. (2014). **The KeY platform for verification and analysis of Java programs**. Dans *VSTTE'14*, volume 8471 de *LNCS*, pages 55–71. Springer.

[Alt-ergo, 2006] Alt-ergo (2006). **The Alt-Ergo SMT solver**. http://alt-ergo.lri.fr.

[Arndt, 2010] Arndt, J. (2010). **Matters Computational - Ideas, Algorithms, Source Code [The fxtbook]**. http://www.jjj.de.

[Arquès et al., 2000] Arquès, D., et Béraud, J.-F. (2000). **Rooted maps on orientable surfaces, Riccati's equation and continued fractions**. *Discrete Mathematics*, 215(1-3) :1–12.

[Ayala-Rincón et al., 2016] Ayala-Rincón, M., Fernández, M., et Rocha-Oliveira, A. C. (2016). **Completeness in PVS of a nominal unification algorithm**. Dans *Electronic Notes in Theoretical Computer Science (LSFA'15)*, volume 323, pages 57 – 74. Elsevier.

[Baril, 2007] Baril, J.-L. (2007). **Gray code for permutations with a fixed number of cycles**. *Discrete Mathematics*, 307(13) :1559 – 1571.

[Baril et al., 2016] Baril, J.-L., Genestier, R., Giorgetti, A., et Petrossian, A. (2016). **Rooted planar maps modulo some patterns**. *Discrete Mathematics*, 339(4) :1199–1205.

[Baril et al., 2015a] Baril, J.-L., et Petrossian, A. (2015a). **Equivalence classes of Dyck paths modulo some statistics**. *Discrete Mathematics*, 338(4) :655 – 660.

[Baril et al., 2015b] Baril, J.-L., et Petrossian, A. (2015b). **Equivalence classes of Motzkin paths modulo a pattern of length at most two**. *Journal of Integer Sequences*, 18 :Article 15.7.1.

[Barnett et al., 2011] Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., et Venter, H. (2011). **Specification and verification : The Spec# experience**. *Commun. ACM*, 54(6) :81–91.

[Barrett et al., 2007] Barrett, C., et Tinelli, C. (2007). **CVC3**. Dans *CAV'07*, volume 4590 de *LNCS*, pages 298–302. Springer.

[Baudin et al., 2015] Baudin, P., Bobot, F., Correnson, L., et Dargaye, Z. (2015). **WP Plug-in Manual, version 0.9 for Magnesium-20151002**. http://frama-c.com/download/wp-manual-Magnesium-20151002.pdf.

[Baudin et al., 2013] Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., et Prevosto, V. (2013). **ACSL : ANSI/ISO C Specification Language**. http://frama-c.com/acsl.html.

[Beckert et al., 2007] Beckert, B., Hähnle, R., et Schmitt, P. H. (2007). **Verification of Object-Oriented Software : The KeY Approach**, volume 4334 de *LNCS*. Springer.

[Bernardi, 2007] Bernardi, O. (2007). **Bijective counting of tree-rooted maps and shuffles of parenthesis systems**. *Electronic Journal of Combinatorics*, 14(1). http://www.combinatorics.org/ojs/index.php/eljc/article/view/v14i1r9/pdf.

[Bertot et al., 2004] Bertot, Y., et Castéran, P. (2004). **Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions**. Texts in theoretical computer science. Springer.

[Bertot et al., 2008] Bertot, Y., Gonthier, G., Ould Biha, S., et Pasca, I. (2008). **Canonical big operators**. Dans *TPHOL'08*, volume 5170 de *LNCS*, pages 86–101. Springer.

[Bertrand, 1998] Bertrand, Y. (1998). **Topofil : un modeleur interactif d'objets 3D à base topologique.** *Technique et Science Informatiques*, 17(4) :443–483.

[Bertrand et al., 1998] Bertrand, Y., et Dufourd, J.-F. (1998). **Du modèle géométrique à la programmation par les spécifications algébriques.** *Technique et Science Informatiques*, 17(4) :485–516.

[Bobot et al., 2013] Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G., et Paskevich, A. (2013). **The Why3 platform 0.81, tutorial and reference manual**. https://hal.inria.fr/hal-00822856.

[Bradley et al., 2006] Bradley, A. R., Manna, Z., et Sipma, H. B. (2006). **What's decidable about arrays ?** Dans *VMCAI'06*, volume 3855 de *LNCS*, pages 427–442. Springer.

[Brun et al., 2012] Brun, C., Dufourd, J.-F., et Magaud, N. (2012). **Designing and proving correct a convex hull algorithm with hypermaps in Coq**. *Comput. Geom.*, 45(8) :436–457.

[Bulwahn, 2012] Bulwahn, L. (2012). **The new Quickcheck for Isabelle**. Dans *CPP'12*, volume 7679 de *LNCS*, pages 92–108. Springer, Heidelberg.

[Carlier et al., 2012] Carlier, M., Dubois, C., et Gotlieb, A. (2012). **A certified constraint solver over finite domains**. Dans *FM'12*, volume 7436 de *LNCS*, pages 116–131. Springer, Heidelberg.

[Claessen et al., 2000] Claessen, K., et Hughes, J. (2000). **QuickCheck : a lightweight tool for random testing of Haskell programs**. Dans *CFP'00*, volume 35 de *SIGPLAN Not.*, pages 268–279. ACM.

[Comtet, 1970] Comtet, L. (1970). **Analyse combinatoire, 1**. Presses universitaires de France.

[Cori, 1975] Cori, R. (1975). **Un code pour les graphes planaires et ses applications**. Société mathématique de France.

[Cori, 2009] Cori, R. (2009). **Indecomposable permutations, hypermaps and labeled dyck paths**. *Journal of Combinatorial Theory, Series A*, 116(8) :1326–1343.

[Cori et al., 1986] Cori, R., Dulucq, S., et Viennot, G. (1986). **Shuffle of parenthesis systems and Baxter permutations**. *Journal of Combinatorial Theory, Series A*, 43(1) :1 – 22.

[Cori et al., 2012] Cori, R., Robson, J. M., et Mathieu, C. (2012). **On the number of indecomposable permutations with a given number of cycles**. *Electronic Journal of Combinatorics*, 19(1) :49. http://www.combinatorics.org/ojs/index.php/eljc/article/view/v19i1p49.

[CVC4, 2012] CVC4 (2012). **CVC4**. http://cvc4.cs.nyu.edu/web/.

[Daca et al., 2016] Daca, P., Henzinger, T. A., et Kupriyanov, A. (2016). **Array folds logic**. *ArXiv e-prints*. http://arxiv.org/abs/1603.06850.

[Dahlweid et al., 2009] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., et Schulte, W. (2009). **Vcc : Contract-based modular verification of concurrent C**. Dans *ICSE'09*, pages 429–430. IEEE Computer Society.

[Damiand, 2010] Damiand, G. (2010). **Contributions aux cartes combinatoires et cartes généralisées : Simplification, modèles, invariants topologiques et applications**. Mémoire d'HDR, https://tel.archives-ouvertes.fr/tel-00538456.

[Damiand et al., 2014] Damiand, G., et Lienhardt, P. (2014). **Combinatorial Maps : Efficient Data Structures for Computer Graphics and Image Processing**. A K Peters/-CRC Press.

[De Moura et al., 2008] De Moura, L., et Bjørner, N. (2008). **Z3 : An efficient SMT solver**. Dans *TACAS'08*, volume 4963 de *LNCS*, pages 337–340. Springer.

[Deutsch, 1999] Deutsch, E. (1999). **Dyck path enumeration**. *Discrete Mathematics*, 204(1–3) :167 – 202. Selected papers in honor of Henry W. Gould.

[Diakite, 2015] Diakite, A. A. (2015). **Application of the combinatiorial maps to geometric and semantic modelling of buildings**. Thèse, Univ. Claude Bernard - Lyon I.

[Dijkstra, 1975] Dijkstra, E. W. (1975). **Guarded commands, nondeterminacy and formal derivation of programs**. *Commun. ACM*, 18(8) :453–457.

[Dubois et al., 2016] Dubois, C., Giorgetti, A., et Genestier, R. (2016). **Tests and proofs for enumerative combinatorics**. Dans *TAP'16*, volume 6792 de *LNCS*, pages 57–75. Springer.

[Dubois et al., 2007] Dubois, C., et Mota, J.-M. (2007). **Geometric modeling with B : formal specification of generalized maps**. *Journal of Scientific & Practical Computing*, 1(2) :9–24.

[Dufourd, 2007] Dufourd, J.-F. (2007). **Design and formal proof of a new optimal image segmentation program with hypermaps**. *Pattern Recogn.*, 40(11) :2974–2993.

[Dufourd, 2008] Dufourd, J.-F. (2008). **Polyhedra genus theorem and Euler formula : A hypermap-formalized intuitionistic proof**. *Theor. Comput. Sci.*, 403(2-3) :133–159.

[Dufourd, 2009] Dufourd, J.-F. (2009). **An intuitionistic proof of a discrete form of the Jordan curve theorem formalized in Coq with combinatorial hypermaps**. *Journal of Automated Reasoning*, 43(1) :19–51.

[Dufourd et al., 2000] Dufourd, J.-F., et Puitg, F. (2000). **Functional specification and prototyping with oriented combinatorial maps**. *Comput. Geom.*, 16(2) :129–156.

[Dutertre, 2014] Dutertre, B. (2014). **Yices 2.2**. Dans *CAV'14*, volume 8559 de *LNCS*, pages 737–744. Springer.

[Enderlin et al., 2011] Enderlin, I., Dadeau, F., Giorgetti, A., et Othman, A. B. (2011). **Praspel : A specification language for contract-based testing in PHP**. Dans *ICTSS'11*, volume 7019 de *LNCS*, pages 64–79. Springer.

[Engel, 2009] Engel, C. (2009). **Sum comprehensions in KeY**. http://i12www.iti.uni-karlsruhe.de/~key/keysymposium09/slides/Sum_comprehensions_in_KeY.pdf.

[Filliâtre et al., 2013] Filliâtre, J.-C., et Paskevich, A. (2013). **Why3 — where programs meet provers**. Dans *ESOP'13*, volume 7792 de *LNCS*, pages 125–128. Springer.

[Filliâtre, 2012] Filliâtre, J.-C. (2012). **Verifying two lines of C with Why3 : An exercise in program verification.** Dans *VSTTE'12*, volume 7152 de *LNCS*, pages 83–97. Springer.

[Floyd, 1967] Floyd, R. W. (1967). **Assigning meanings to programs**. Dans *Symposia in Applied Mathematics*, volume 19, pages 19–32. Springer.

[Genestier, 2016] Genestier, R. (2016). **Vérification formelle de programmes de génération de données structurées**. Dans *Approches Formelles dans l'Assistance au Développement Logiciel (AFADL'16)*, pages 67–71. http://events.femto-st.fr/sites/femto-st.fr.gdr-gpl-2016/files/content/AFADL-2016.pdf.

[Genestier et al., 2016] Genestier, R., et Giorgetti, A. (2016). **Spécification et vérification formelle d'opérations sur les permutations**. Dans *AFADL'16*, pages 72–78. http://events.femto-st.fr/sites/femto-st.fr.gdr-gpl-2016/files/content/AFADL-2016.pdf.

[Genestier et al., 2015a] Genestier, R., Giorgetti, A., et Petiot, G. (2015a). **Gagnez sur tous les tableaux**. Dans *JFLA'15*. https://hal.inria.fr/hal-01099135.

[Genestier et al., 2015b] Genestier, R., Giorgetti, A., et Petiot, G. (2015b). **Sequential generation of structured arrays and its deductive verification**. Dans *TAP'15*, volume 9154 de *LNCS*, pages 109–128. Springer.

[Giorgetti et al., 2012] Giorgetti, A., et Senni, V. (2012). **Specification and validation of algorithms generating planar Lehman words**. https://hal.inria.fr/hal-00753008.

[Gligoric et al., 2010] Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., et Marinov, D. (2010). **Test generation through programming in UDITA**. Dans *ICSE'10*, volume 1, pages 225–234. ACM.

[Gonthier, 2005] Gonthier, G. (2005). **A computer checked proof of the Four Colour Theorem**. http://research.microsoft.com/gonthier/4colproof.pdf.

[Gonthier, 2008] Gonthier, G. (2008). **The Four Colour Theorem : Engineering of a Formal Proof**. Dans *Computer Mathematics (ASCM'07)*, volume 5081 de *LNCS*, pages 333–333, Berlin, Heidelberg. Springer.

[Gonthier et al., 2015] Gonthier, G., Mahboubi, A., et Tassi, E. (2015). **A Small Scale Reflection Extension for the Coq system**. Research Report RR-6455, Inria Saclay Ile de France. https://hal.inria.fr/inria-00258384.

[Guitter, 2015] Guitter, E. (2015). **The distance-dependent two-point function of quadrangulations : a new derivation by direct recursion**. *ArXiv e-prints*. https://arxiv.org/abs/1512.00179.

[Hatcliff et al., 2012] Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., et Parkinson, M. (2012). **Behavioral interface specification languages**. *ACM Comput. Surv.*, 44(3) :1–67.

[Hoare, 1969] Hoare, C. A. R. (1969). **An axiomatic basis for computer programming**. *Commun. ACM*, 12(10) :576–580.

[Hritcu et al., 2015] Hritcu, C., Lampropoulos, L., Dénès, M., et Paraskevopoulou, Z. (2015). **Randomized property-based testing plugin for Coq**. https://github.com/QuickChick.

[Jackson et al., 1996] Jackson, D., et Damon, C. (1996). **Elements of style : Analyzing a software design feature with a counterexample detector**. *IEEE Transactions on Software Engineering*, 22(7) :484–495.

[Jacquot, 2014] Jacquot, A. (2014). **Graft reconstruction, analytic combinatorics and analytical generation**. Thèse, Univ. Paris-Nord - Paris XIII.

[Kaufmann et al., 2004] Kaufmann, M., et Moore, J. S. (2004). **The ACL2 home page**. http://www.cs.utexas.edu/users/moore/acl2/.

[Kirchner et al., 2012] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., et Yakobowski, B. (2012). **Frama-C : a software analysis perspective**. *Formal Aspects of Computing*, 27(3) :573–609.

[Kitaev, 2011] Kitaev, S. (2011). **Patterns in permutations and words.** Springer.

[Knuth, 1997] Knuth, D. E. (1997). **The art of computer programming, volume 1 (3rd ed.) : fundamental algorithms**. Addison Wesley Longman Publishing Co., Inc.

[Kovács et al., 2009] Kovács, L., et Voronkov, A. (2009). **Finding loop invariants for programs over arrays using a theorem prover**. Dans *FASE'09*, volume 5503 de *LNCS*, pages 470–485. Springer Berlin.

[Kraemer et al., 2013] Kraemer, P., Untereiner, L., Jund, T., Thery, S., et Cazier, D. (2013). **CGoGN : n-dimensional meshes with combinatorial maps**. Dans *Proceedings of the 22nd International Meshing Roundtable, IMR'13*, pages 485–503. Springer.

[Kreher et al., 1999] Kreher, D. L., et Stinson, D. R. (1999). **Combinatorial algorithms : generation, enumeration, and search**. CRC Press.

[Lando et al., 2004] Lando, S. K., et Zvonkin, A. K. (2004). **Graphs on surfaces and their applications**. Springer.

[Lazarus, 2014] Lazarus, F. (2014). **Combinatorial graphs and surfaces from the computational and topological viewpoint followed by some notes on the isometric embedding of the square flat torus**. Mémoire d'HDR, http://www.gipsa-lab.grenoble-inp.fr/~francis.lazarus/Documents/hdr-Lazarus.pdf.

[Leino, 2010] Leino, K. R. M. (2010). **Dafny : An automatic program verifier for functional correctness**. Dans *LPAR'10*, volume 6355 de *LNCS*, pages 348–370. Springer Berlin Heidelberg.

[Leroy, 2009] Leroy, X. (2009). **Formal verification of a realistic compiler**. *Commun. ACM*, 52(7) :107–115.

[Lienhardt, 1991] Lienhardt, P. (1991). **Topological models for boundary representation : a comparison with n-dimensional generalized maps**. *Computer-Aided Design*, 23(1) :59 – 82.

[Liskov et al., 1986] Liskov, B., et Guttag, J. (1986). **Abstraction and Specification in Program Development**. MIT Press, Cambridge, MA, USA.

[Marinov et al., 2001] Marinov, D., et Khurshid, S. (2001). **TestEra : A novel framework for automated testing of Java programs**. Dans *ASE'01*, pages 22–31. IEEE Computer Society.

[Mathematical Components team, 2015] Mathematical Components team (2006-2015). **Library mathcomp.ssreflect.fingraph**. http://math-comp.github.io/math-comp/htmldoc/mathcomp.ssreflect.fingraph.html.

[Meng et al., 2008] Meng, J., et Paulson, L. C. (2008). **Translating higher-order clauses to first-order clauses.** *Journal of Automated Reasoning*, 40(1) :35–60.

[Meyer et al., 1987] Meyer, B., Nerson, J.-M., et Matsuo, M. (1987). **EIFFEL : Object-oriented design for software engineering**. Dans *ESEC'87*, volume 289 de *LNCS*, pages 221–229. Springer.

[Mullin, 1967] Mullin, R. C. (1967). **On the enumeration of tree-rooted maps**. *Canad. J. Math.*, 19 :174–183.

[Oe et al., 2012] Oe, D., Stump, A., Oliver, C., et Clancy, K. (2012). **Versat : A verified modern SAT solver**. Dans *VMCAI'12*, volume 7148 de *LNCS*, pages 363–378. Springer.

[Ossona de Mendez et al., 2004] Ossona de Mendez, P., et Rosenstiehl, P. (2004). **Transitivity and connectivity of permutations**. *Combinatorica*, 24(3) :487–501.

[Ossona De Mendez et al., 2006] Ossona De Mendez, P., et Rosenstiehl, P. (2006). **Encoding pointed maps by double occurrence words**, pages 701–712. Eptalofos.

[Owre et al., 1996] Owre, S., Rajan, S., Rushby, J., Shankar, N., et Srivas, M. (1996). **PVS : Combining specification, proof checking, and model checking**. Dans *CAV'96*, volume 1102 de *LNCS*, pages 411–414. Springer.

[Paraskevopoulou et al., 2015] Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., et Pierce, B. C. (2015). **Foundational property-based testing**. Dans *ITP'15*, volume 9236 de *LNCS*, pages 325–343. Springer.

[Paulson, 1993] Paulson, L. C. (1993). **Introduction to Isabelle**. Rapport technique UCAM-CL-TR-280, University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-280.dvi.gz.

[Petiot et al., 2014] Petiot, G., Kosmatov, N., Giorgetti, A., et Julliand, J. (2014). **How test generation helps software specification and deductive verification in Frama-C**. Dans *TAP'14*, volume 8570 de *LNCS*, pages 204–211. Springer.

[Planat et al., 2015] Planat, M., Giorgetti, A., Holweck, F., et Saniga, M. (2015). **Quantum contextual finite geometries from dessins d'enfants**. *Int. J. of Geometric Methods in Modern Physics*, 12(7) :18 pages.

[Prissette, 2010] Prissette, C. (2010). **An algorithm to list all the fixed-point free involutions on a finite set**. *ArXiv e-prints*. http://arxiv.org/abs/1006.3993.

[Pugh, 1991] Pugh, W. (1991). **The Omega test : A fast and practical integer programming algorithm for dependence analysis**. Dans *Supercomputing'91*, pages 4–13. ACM.

[Reynolds et al., 2016] Reynolds, A., et Blanchette, J. C. (2016). **A decision procedure for (co)datatypes in SMT solvers**. *Journal of Automated Reasoning*, pages 1–22.

[Runciman et al., 2008] Runciman, C., Naylor, M., et Lindblad, F. (2008). **Smallcheck and lazy smallcheck : automatic exhaustive testing for small values**. Dans *Haskell'08*, pages 37–48. ACM.

[Ruskey, 2003] Ruskey, F. (2003). **Combinatorial Generation**. http://www.1stworks.com/ref/ruskeycombgen.pdf.

[Seidel et al., 2014] Seidel, E. L., Vazou, N., et Jhala, R. (2014). **Type targeted testing**. *ArXiv e-prints*. http://arxiv.org/abs/1410.5370.

[Senni, 2012] Senni, V. (2012). **Validation library**. https://subversion.assembla.com/svn/validation/.

[Senni et al., 2012] Senni, V., et Fioravanti, F. (2012). **Generation of test data structures using constraint logic programming**. Dans *TAP'12*, volume 7305 de *LNCS*, pages 115–131. Springer.

[Stanley, 1997] Stanley, R. (1997). **Enumerative Combinatorics**, volume 1. Cambridge University Press. http://www-math.mit.edu/~rstan/ec/ec1.pdf.

[Sullivan et al., 2004] Sullivan, K., Yang, J., Coppit, D., Khurshid, S., et Jackson, D. (2004). **Software assurance by bounded exhaustive testing**. *SIGSOFT Softw. Eng. Notes*, 29(4) :133–142.

[The OEIS Foundation Inc., 2010] The OEIS Foundation Inc. (2010). **The On-Line Encyclopedia of Integer Sequences**. http://oeis.org.

[Tushkanova et al., 2010] Tushkanova, E., Giorgetti, A., Marché, C., et Kouchnarenko, O. (2010). **Specifying generic Java programs : two case studies**. Dans *LDTA'10*, pages 92–106. ACM.

[Tutte, 1963] Tutte, W. T. (1963). **A census of planar maps**. *Canad. J. Math.*, 15 :249–271.

[Tutte, 1968] Tutte, W. T. (1968). **On the enumeration of planar maps**. *Bull. Amer. Math. Soc.*, 74 :64–74.

[Tutte, 1971] Tutte, W. T. (1971). **What is a map ?** *New directions in the theory of graphs*, pages 309–325.

[Tutte, 1979] Tutte, W. T. (1979). **Combinatorial oriented maps**. *Canad. J. Math.*, 31(5) :986–1004.

[Vajnovszki, 2010] Vajnovszki, V. (2010). **Generating involutions, derangements, and relatives by ECO**. *Discrete Mathematics and Theoretical Computer Science*, 12(1) :109–122.

[Vidal, 2010] Vidal, S. (2010). **Groupe modulaire et cartes combinatoires : génération et comptage**. Thèse, Univ. Lille. http://www.theses.fr/2010LIL10180.

[Walsh, 2001] Walsh, T. (2001). **Gray codes for involutions**. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 36 :95–118.

[Walsh, 1971] Walsh, T. R. S. (1971). **Combinatorial enumeration of non-planar maps**. Thèse, Univ. Toronto.

[Walsh et al., 1972a] Walsh, T. R. S., et Lehman, A. B. (1972a). **Counting rooted maps by genus I**. *Journal of Combinatorial Theory, Series B*, 13 :192–218.

[Walsh et al., 1972b] Walsh, T. R. S., et Lehman, A. B. (1972b). **Counting rooted maps by genus ii**. *Journal of Combinatorial Theory, Series B*, 13(2) :122 – 141.

[Weber, 2011] Weber, T. (2011). **SMT solvers : New oracles for the HOL theorem prover**. *Int. J. Softw. Tools Technol. Transf.*, 13(5) :419–429.

[Williams, 2010] Williams, N. (2010). **Abstract path testing with PathCrawler**. Dans *AST'10*, pages 35–42. ACM.

[Zito, 2014] Zito, A. (2014). **quickcheck4c : A QuickCheck for C**. https://github.com/nivox/quickcheck4c.

Document réalisé avec LaTeX et :
le style LaTeX pour Thèse de Doctorat créé par S. Galland — http://www.multiagent.fr/ThesisStyle
la collection de paquets tex-upmethodology — http://www.arakhne.org/tex-upmethodology/

Résumé :

Le problème général de la preuve de propriétés de programmes impératifs est indécidable. Pour des langages de programmation et de propriétés plus restrictifs, des sous-problèmes décidables sont connus. En pratique, grâce à des heuristiques, les outils de preuve de programmes automatisent des preuves qui sortent du cadre théorique de ces sous-problèmes décidables connus. Nous illustrons cette réussite pratique en construisant un catalogue de preuves, pour des programmes et des propriétés de nature similaire et de complexité croissante. Ces programmes sont principalement des générateurs de cartes combinatoires.
Ainsi, ce travail contribue aux domaines de recherche de la combinatoire énumérative et du génie logiciel. Nous distribuons une bibliothèque C de générateurs exhaustifs bornés de tableaux structurés, formellement spécifiés en ACSL et vérifiés avec le greffon WP de la plateforme d'analyse Frama-C. Nous proposons également une méthodologie de test qui facilite la preuve interactive en Coq, une étude formelle des cartes originale, et de nouveaux résultats en combinatoire énumérative.

Mots-clés : Programmes impératifs, preuve de programmes, langage C, ACSL, Frama-C, Coq, cartes combinatoires

Abstract:

The general problem of proving properties of imperative programs is undecidable. Some subproblems – restricting the languages of programs and properties – are known to be decidable. In practice, thanks to heuristics, program proving tools sometimes automate proofs for programs and properties living outside of the theoretical framework of known decidability results. We illustrate this fact by building a catalog of proofs, for similar programs and properties of increasing complexity. Most of these programs are combinatorial map generators.
Thus, this work contributes to the research fields of enumerative combinatorics and software engineering. We distribute a C library of bounded exhaustive generators of structured arrays, formally specified in ACSL and verified with the WP plugin of the Frama-C analysis platform. We also propose a testing-based methodology to assist interactive proof in Coq, an original formal study of maps, and new results in enumerative combinatorics.

Keywords: Imperative programs, program proof, C language, ACSL, Frama-C, Coq, combinatorial maps

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC UNIVERSITÉ DE FRANCHE-COMTÉ