UNIVERSITÉ DU QUÉBEC À MONTRÉAL

LES CODES GRAY POUR LES IDÉAUX D'UN POSET ET POUR D'AUTRES OBJETS COMBINATOIRES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES
CONCENTRATION EN INFORMATIQUE MATHÉMATIQUE

PAR

MOHAMED ABDO

FEVRIER 2006

REMERCIEMENTS

Je tiens à remercier Timothy Walsh, mon directeur de recherche, pour ce qu'il m'a appris, pour ses encouragements, pour le temps qu'il m'a consacré en répondant à mes questions, pour les conseils qu'il m'a donnés quand j'étais démonstrateur de son cours, et pour son appui financier. Sans son encadrement, ce travail n'aurait pas été terminé.

Je tiens aussi à remercier Mme Odile Marcotte pour ce qu'elle m'a appris et pour son intérêt envers les étudiants, et surtout, pour ses précieux conseils, ses encouragements et sa collaboration.

Je remercie le laboratoire de combinatoire et d'informatique mathématique (LaCIM), représenté par Pierre Leroux et Christophe Reutenauer, pour le soutien qu'il m'a accordé.

Je remercie Geneviève Paquin d'avoir accepté de corriger les fautes de grammaire.

Merci à Manon Gauthier, qui fournit tous les conseils administratifs aux étudiants et qui leur porte beaucoup d'intérêt. Merci à Gisèle Legault, qui, avec ses connaissances informatiques, m'a permis de sauver beaucoup de temps. Merci à Lise Tourigny et à André Lauzon pour leur coopération et leurs réponses à mes questions, ainsi qu'à tout le personnel des départements de mathématiques et d'informatique.

Enfin, je remercie mes parents qui m'ont soutenu et encouragé constamment à réaliser mes rêves dans les études.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Pruesse et Ruskey ont trouvé un code Gray pour les idéaux d'un ensemble partiellement ordonné (poset) et un algorithme récursif pour les engendrer. Dans ce mémoire, un algorithme non-récursif qui engendre la même liste d'idéaux est présenté. De plus, plusieurs autres codes Gray classiques majoritairement reliés aux posets et leurs implantations sont étudiés. Plus particulièrement, les codes Gray de Chase et de Ruskey pour les combinaisons, celui de Ruskey et Proskurowski pour les mots de Dyck et celui de Walsh pour les involutions sans point fixe sont étudiés. Le code Gray de Chase est présenté sous forme d'un programme FORTRAN. Vajnovszki et Walsh ont trouvé une implantation plus simple sans en donner une preuve formelle ; une telle preuve est présentée dans ce mémoire.

MOTS-CLÉS : Code Gray, idéal, ensemble partiellement ordonné (poset), extension linéaire, poset forêt, algorithme, non-récursif, sans-boucle, temps constant amorti (CAT).

INTRODUCTION

L'un des premiers problèmes posés dans le domaine des algorithmes combinatoires est celui de générer efficacement les éléments d'une classe combinatoire de façon à ce que chaque élément soit généré une seule fois. En pratique, beaucoup de problèmes requièrent pour leurs solutions le tirage d'un objet aléatoire d'une classe combinatoire ou pire, une recherche exhaustive parmi tous les objets d'une classe. Bien qu'au début, le travail en combinatoire se concentrait sur le dénombrement, dans les années 1960, il était déjà possible à l'aide de l'ordinateur de générer les objets des classes combinatoires. Pourtant, afin que cette génération soit réalisable même pour des objets de tailles modérées, la méthode combinatoire de génération doit être extrêmement efficace.

L'approche choisie était d'essayer de générer les objets dans une liste telle que les objets successifs diffèrent peu (par un changement borné indépendamment de la taille de l'objet).

L'exemple classique est le *binary reflected Gray code* (code Gray binaire reflété) (Gilbert, 58; Gray, 53). Il consiste à générer toutes les chaînes binaires de longueur n de façon à ce que les chaînes successives diffèrent d'une seule position.

L'avantage anticipé d'une telle approche est double.

- D'abord, la génération d'objets successifs peut être rapide. Bien que pour beaucoup de familles combinatoires, l'algorithme pour générer les objets en ordre lexicographique requiert en moyenne un temps constant par élément, pour d'autres familles, comme les extensions linéaires, une telle performance n'a pu être obtenue que par l'approche de code Gray (Pruesse et Ruskey, 94).
- D'autre part, pour les applications connues, il est probable que les objets combinatoires qui diffèrent par un changement borné indépendamment de leurs tailles soient associés à des solutions qui diffèrent seulement par une petite quantité de calcul.

Par exemple, Nijenhuis et Wilf montrent comment utiliser le code Gray pour accélérer le calcul de permanent (Nijenhuis et Wilf, 78). Le permanent est l'analogue du déterminant où tous les signes dans l'expansion en sous-détermiants sont positifs. Mis à part la considération du calcul, des questions ouvertes dans divers domaines mathématiques peuvent être posées comme problèmes de code Gray.

Enfin, l'un des principaux atouts dans le domaine est l'élégance des constructions récursives qui sont impliquées qui donne une nouvelle vue à l'intérieur de la structure des familles combinatoires.

Le terme de **code Gray** est apparu pour la première fois dans (Joichi, White et Williamson, 80) et est maintenant utilisé pour désigner n'importe quelle méthode de génération d'objets combinatoires où les objets successifs diffèrent d'une façon prédéfinie par un petit changement.

Pourtant, les origines de la génération des objets avec un changement minimal se trouvent dès le premier travail de Gray : (Gray, 53; Wells, 61; Trotter, 62; Johnson, 63; Lehmer, 65; Chase, 70; Ehrlich, 73; Nijenhuis et Wilf, 78).

Dans son article sur les origines du code Gray binaire, Heath décrit un télégraphe inventé par Emile Baudot en 1878 qui a utilisé le *binary reflected Gray code* (Heath, 72). Selon Heath, Baudot aurait reçu une médaille d'or pour son télégraphe à l'exposition universelle de Paris en 1878, comme Thomas Edison et Alexander Graham Bell.

Comme exemples de code Gray, nous trouvons entre autres :

1. Générer toutes les permutations de $1, 2, \ldots, n$ de façon à ce que les permutations successives diffèrent par l'échange d'une paire d'éléments adjacents (Johnson, 63; Trotter, 62).
2. Générer tous les k-sous-ensembles d'un n-ensemble de façon à ce que deux sous-ensembles successifs diffèrent par un seul élément (Bitner, Ehrlich et Reingold, 76; Buck et Wiedemann, 84; Eades, Hickey, et Read, 84; Eades et McKay, 84; Nijenhuis et Wilf, 78; Ruskey, 88).

3. Générer tous les arbres binaires de façon à ce que deux arbres consécutifs diffèrent par une rotation d'un seul noeud (Lucas, 87; Lucas, Roelants et Ruskey, 93).
4. Générer tous les arbres de recouvrement d'un graphe de façon à ce que deux arbres successifs diffèrent par une seule arête (Holzmann et Harary, 72; Cummings, 66).
5. Générer toutes les partitions d'un entier naturel n de façon à ce que dans les partitions successives, une partie est incrémentée par 1 et une partie est décrémentée par 1 (Savage, 89).
6. Générer les extensions linéaires de certains posets de façon à ce que les éléments successifs diffèrent seulement par une transposition (Ruskey, 92; Pruesse et Ruskey, 91; Stachowiak, 92; West, 93).
7. Générer les éléments d'un groupe de Coxeter de façon à ce que les éléments successifs diffèrent par une réflexion (Conway, Sloane et Wilks, 89).
8. Générer toutes les chaînes de parenthèses bien formées, de façon à ce que les chaînes successives diffèrent par une transposition de deux lettres (Proskurowski et Ruskey, 90). Walsh a été le premier à trouver un algorithme sans boucle pour ce problème (Walsh, 98).
9. Générer toutes les involutions de longueur n de façon à ce qu'une involution soit obtenue de son prédécesseur par une ou deux transpositions, ou une rotation de trois éléments (Walsh, 2001).

Les applications des codes Gray se trouvent dans différents domaines : test de circuit (Robinson et Cohn, 81), codage de signal (Ludman, 81), tri de documents sur des rayons (ordering of documents on shelves) (Losee, 92), compression de données (Richard, 86), statistiques (Diaconis et Holmes, 94), traitement d'images et graphes (Amalraj, Sundararajan et Dhar, 90), allocation de processus dans l'hypercube (Chen et Shin, 90), hachage (Faloutsos, 88), calcul de permanent (Nijenhuis et Wilf, 78), stockage et recherche d'information (Chang, Chen et Chen, 92) et les jeux casse-tête comme les anneaux chinois et les tours de Hanoi (Gardner, 72).

Une variation des codes Gray est la génération d'objets de telle sorte que la différence entre deux objets successifs, bien que fixée, n'a pas à être minimale. Comme exemple, nous trouvons le problème de la génération de toutes les permutations de $1, 2, \ldots, n$ de telle sorte que les permutations successives diffèrent en toute position (Wilf, 89).

Le problème de la génération d'un exemplaire de chacun des objets d'une classe combinatoire telle que les objets successifs diffèrent d'une manière prédéfinie, peut être formulé comme un problème du chemin/cycle hamiltonien : les sommets du graphe sont les objets eux-mêmes et une arête relie deux objets s'ils diffèrent de la manière prédéfinie. Ce graphe possède un chemin hamiltonien si et seulement si la liste d'objets en question existe.
Un cycle hamiltonien correspond à un chemin hamiltonien dans lequel le premier et le dernier objets diffèrent de la même manière prédéfinie. Or, comme le problème de déterminer si un graphe possède un chemin/cycle hamiltonien est *NP*-complet (Garey et Johnson, 79), il n'y a pas d'algorithme général efficace pour découvrir des codes Gray combinatoires.

Néanmoins, dans les problèmes de codes Gray, le graphe associé possède très fréquemment une certaine symétrie. En particulier, il peut appartenir à la classe des graphes sommet-transitifs. Un graphe G est dit sommet-transitif si pour n'importe quelle paire de sommets u, v de G, il y a un automorphisme $\emptyset$ sur G avec $\emptyset(u) = v$.

Bien qu'il semble que beaucoup de codes Gray requièrent des stratégies adaptées au problème en main, quelques techniques générales et structures unifiées ont émergé. L'article (Joichi, White et Williamson, 80) considère des familles d'objets combinatoires dont la taille est définie par une récurrence de forme particulière et quelques résultats généraux sont obtenus pour la construction de codes Gray pour ces familles. Ruskey (Ruskey, 92) a montré que certains codes Gray pour générer des objets combinatoires peuvent être vus comme des cas particuliers de la génération des extensions linéaires d'un poset associé de telle sorte que les extensions successives diffèrent par une transposition.

Walsh a généralisé la méthode de Chase (Chase, 89) pour n'importe quelle liste de mots dans laquelle tous les mots possédant un même suffixe forment un intervalle de mots consécutifs (Walsh, 95). De plus, il a trouvé des conditions suffisantes sur un code Gray pour que la méthode d'Ehrlich (Ehrlich, 73) possède une implantation sans boucle et il a généralisé cette méthode pour qu'elle fonctionne sous des conditions moins restrictives (Walsh, 2000).

Ainsi, beaucoup de problèmes intéressants en combinatoire, théorie des graphes, théorie des groupes et calcul par ordinateur ainsi que quelques problèmes ouverts sont posés comme problèmes de codes Gray (Savage, 97).
Dans ce mémoire, nous allons aborder plusieurs problèmes de codes Gray, quelques-uns en détail, et nous allons nous concentrer sur les idéaux d'un poset.

CHAPITRE I

CODE GRAY POUR LES IDÉAUX D'UN POSET (ENSEMBLE PARTIELLEMENT ORDONNÉ)

1.1 Introduction

Pruesse et Ruskey ont trouvé un code Gray pour les idéaux d'un poset (Pruesse et Ruskey, 93). Ils ont donné un théorème qui trouve ce code de façon récursive. Ruskey s'est inspiré de ce théorème et a écrit un programme en C pour engendrer récursivement tous les idéaux d'un poset. Nous avons modifié son programme pour engendrer itérativement ces idéaux. Notre implantation a la même complexité asymptotique que l'implantation récursive de Pruesse et Ruskey, mais elle s'exécute un peu plus rapidement.

1.2 Définitions et exemples

Définition 1.1 *Un* ***système d'ensembles*** *sur un ensemble fini E est une paire $(E, \mathcal{F})$, où $\mathcal{F} \subseteq 2^E$.*

Définition 1.2 *Un système d'ensembles est un* ***antimatroïde*** *s'il respecte les deux propriétés suivantes :*

1. *si $F \in \mathcal{F}$ avec $F \neq \emptyset$, alors il existe un $x \in F$ tel que $F \setminus \{x\} \in \mathcal{F}$;*
2. *si $A, B \in \mathcal{F}$, alors $A \cup B \in \mathcal{F}$.*

Remarque. Pour simplifier la notation, nous notons $\mathcal{F}$ l'antimatroïde $(E, \mathcal{F})$.

Définition 1.3 *Si $\mathcal{F}$ est un antimatroïde et $A \in \mathcal{F}$, alors nous pouvons obtenir deux nouveaux antimatroïdes par* **contraction** *et par* **restriction** *(deletion) respectivement comme suit :*

$\mathcal{F}/A = \{F \in 2^E : F \cap A = \emptyset \quad et \quad F \cup A \in \mathcal{F}\}$,

$\mathcal{F} \setminus A = \{F \in \mathcal{F} : F \cap A = \emptyset\}$.

Notons que ces deux antimatroïdes induisent une partition des ensembles de $\mathcal{F}$ si $|A| = 1$. Pour alléger la notation, l'ensemble $\{a, b, c\} \in \mathcal{F}$ sera noté abc.

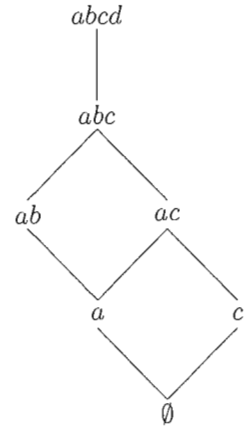

Fig. 1.1 Diagramme de Hasse de $\mathcal{F}$.

Exemple 1.1 *Soient $E = \{a, b, c, d\}$*

et $\mathcal{F} = \{\emptyset, a, c, ab, ac, abc, abcd\}$ (voir la Fig. 1.1).

Alors :

$\mathcal{F}/\{a\} = \{\emptyset, b, c, bc, bcd\}$,

$\mathcal{F} \setminus \{a\} = \{\emptyset, c\}$.

Il semble que la classe la plus importante des antimatroïdes est l'antimatroïde d'un poset (Pruesse et Ruskey, 93).

Définition 1.4 *Un* **poset** *(partially ordered set), ensemble partiellement ordonné, $\mathcal{P}$ est un ensemble d'éléments $S(\mathcal{P})$ muni d'une relation réflexive, transitive et antisymétrique $R(\mathcal{P})$.*

Notation 1.1 *Une* **paire ordonnée** *$(a, b) \in R(\mathcal{P})$ est notée : $a \preceq_{\mathcal{P}} b$ ou plus simplement $a \preceq b$, s'il n'y a pas de confusion.*
Par $a \prec b$, nous voulons dire $a \preceq b$ et $a \neq b$.

Définition 1.5 *Un sous-ensemble $\mathrm{I} \subseteq S(\mathcal{P})$ est un* **idéal** *si ($a \preceq_{\mathcal{P}} b$ et $b \in \mathrm{I}$) implique $a \in \mathrm{I}$.*

Définition 1.6 *Le système d'ensembles $(E, \mathcal{F})$ est un* **antimatroïde d'un poset** *si E est l'ensemble d'éléments d'un poset $\mathcal{P}$ et $\mathcal{F}$ est l'ensemble des idéaux de $\mathcal{P}$.*

Définition 1.7 *Il y a deux sommes standards pour les antimatroïdes. Soient $(E_1, \mathcal{F}_1)$ et $(E_2, \mathcal{F}_2)$ deux antimatroïdes sur deux ensembles différents E_1 et E_2. Prenons $E_1 \cup E_2$ comme ensemble de base. Alors la* **somme directe** *$\mathcal{F}_1 + \mathcal{F}_2$ est l'antimatroïde*
$\mathcal{F}_1 + \mathcal{F}_2 = \{X_1 \cup X_2 : X_1 \in \mathcal{F}_1$ et $X_2 \in \mathcal{F}_2\}$,
et la **somme ordonnée** *$\mathcal{F}_1 \oplus \mathcal{F}_2$ est l'antimatroïde*
$\mathcal{F}_1 \oplus \mathcal{F}_2 = \mathcal{F}_1 \cup \{E_1 \cup X_2 : X_2 \in \mathcal{F}_2\}$.

Définition 1.8 *À partir d'éléments isolés, et en appliquant les opérations de somme directe et ordonnée, nous obtenons l'antimatroïde des* **séries-parallèles** *de posets.*

Exemple 1.2 *Le graphe de Hasse de l'antimatroïde de posets :*

$$1 \oplus (2 + 3) \oplus 4 \oplus (5 + 6) \oplus 7$$

est l'antimatroïde des séries-parallèles de posets de la Fig. 1.2.

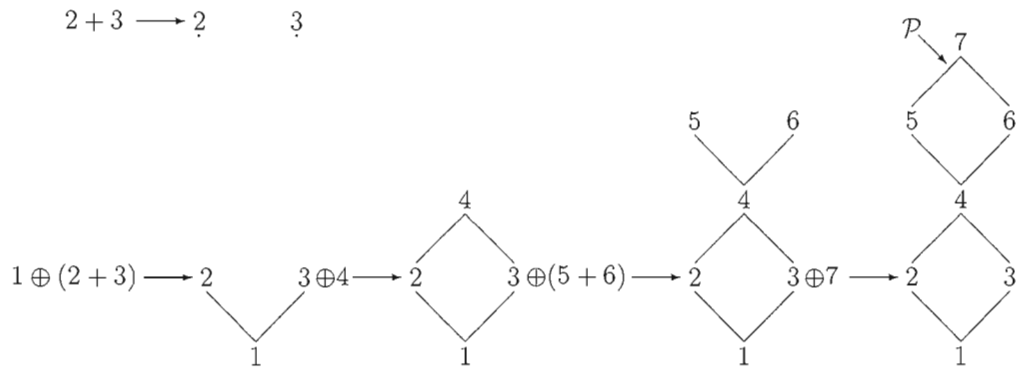

Fig. 1.2 Poset obtenu par sommes directes et ordonnées.

Notation 1.2 *Pour l'antimatroïde $(E, \mathcal{F})$, notons $J(\mathcal{F})$ le graphe non-orienté dont les sommets sont les ensembles de $\mathcal{F}$ ordonnés par inclusion. $J(\mathcal{F})$ est connexe et biparti.*

Définition 1.9 *Le **carré** d'un graphe G est le graphe G^2 dont l'ensemble de sommets est le même que celui de G et est tel que deux sommets sont adjacents dans G^2 s'ils sont adjacents dans G ou s'il y a un autre sommet adjacent à l'un et à l'autre dans G. (Pruesse et Ruskey, 93)*

Définition 1.10 *Un **chemin hamiltonien** dans un graphe est un chemin qui passe une fois et une seule par chaque sommet du graphe.*

Pour certains $\mathcal{F}$, $J(\mathcal{F})$ ne possède pas de chemin hamiltonien même si les parties de $J(\mathcal{F})$ sont de la même cardinalité (voir la Fig. 1.3).

1.3 Algorithme récursif

Lemme 1.1 *Si G est biparti et $G \times K_2$ est hamiltonien, alors G^2 est hamiltonien (Pruesse et Ruskey, 94).*

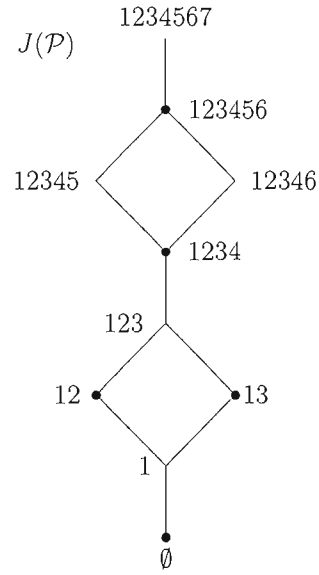

Fig. 1.3 $J(\mathcal{P})$ de l'exemple 1.2 est biparti et les deux parties ont la même cardinalité, mais $J(\mathcal{P})$ ne possède pas de chemin hamiltonien.

Pruesse et Ruskey ont démontré que $J(\mathcal{F}) \times K_2$ est hamiltonien (Pruesse et Ruskey, 93), ce qui revient à dire que les idéaux d'un antimatroïde d'un poset peuvent être générés de telle sorte que deux idéaux successifs (sauf le premier et le dernier) diffèrent par 1 ou 2 éléments selon le lemme 1.1. Considérons le graphe $G \times K_2$, i.e. le produit d'un graphe par une arête. Autrement dit, nous avons deux copies de G, que nous notons $+G$ et $-G$, avec des arêtes reliant les sommets correspondants dans les deux copies. Étant donné un sommet v dans G, nous notons $+v$ et $-v$, les sommets correspondants dans $G \times K_2$ (voir la Fig. 1.4).

Définition 1.11 *Un élément a est un **atome** si $\{a\} \in \mathcal{F}$.*

Définition 1.12 *Un **cycle hamiltonien** dans un graphe est un cycle qui passe une fois et une seule par chaque sommet.*

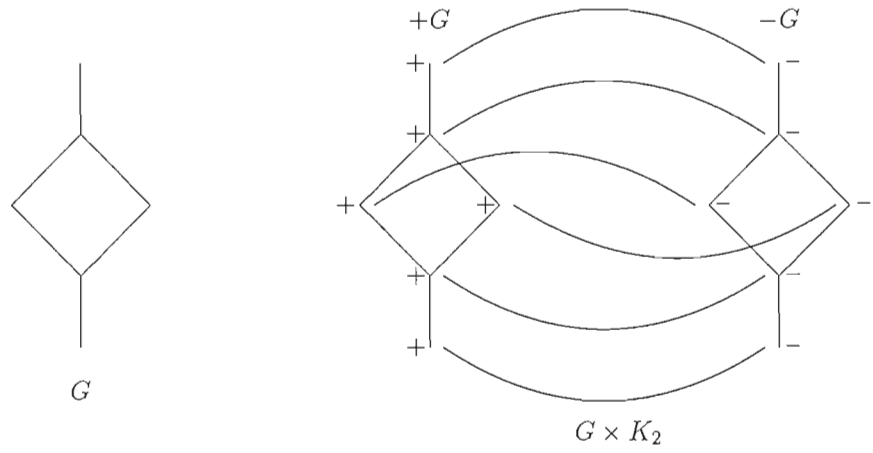

Fig. 1.4 Graphes G et $G \times K_2$.

Ruskey s'est inspiré du théorème 1.1 suivant pour engendrer récursivement tous les idéaux d'un poset.

Théorème 1.1 *(Pruesse et Ruskey, 93) Pour chaque antimatroïde $\mathcal{F}$ et chaque atome x de $\mathcal{F}$, le graphe $J(\mathcal{F}) \times K_2$ possède un cycle hamiltonien incluant les arêtes $[+\emptyset, -\emptyset]$ et $[+\emptyset, +\{x\}]$.*

Preuve : Nous allons prouver ce théorème par récurrence sur $n = |E|$.
Si $n = 1$, alors $J(\mathcal{F}) \times K_2$ est le 4-cycle $+\emptyset, +\{x\}, -\{x\}, -\emptyset$, où $E = \{x\}$.
Si $n > 1$, alors $\{x\} \in \mathcal{F}$ et considérons l'antimatroïde $\mathcal{F}/\{x\}$. Nous avons deux cas selon le nombre de singletons dans $\mathcal{F}$.

1. Si x est le seul atome dans $\mathcal{F}$, alors selon l'hypothèse de récurrence, nous avons un cycle hamiltonien $+\emptyset = X_1, X_2, \ldots, X_p = -\emptyset$ dans $J(\mathcal{F}/x) \times K_2$. Ce cycle peut être étendu au cycle : $+\emptyset, X_1 \cup \{x\}, X_2 \cup \{x\}, \ldots, X_p \cup \{x\}, -\emptyset$ dans $J(\mathcal{F}) \times K_2$; nous voyons clairement que ce cycle contient les arêtes $[+\emptyset, -\emptyset]$ et $[+\emptyset, +\{x\}]$.
2. S'il y a au moins deux singletons $\{x\}$ et $\{y\}$ dans $\mathcal{F}$, alors selon l'hypothèse de récurrence, nous avons les deux cycles hamiltoniens suivants :
   $+\emptyset = X_1, X_2, \ldots, X_p = -\emptyset$ dans $J(\mathcal{F}/x) \times K_2$

et $+\emptyset = Y_1, Y_2, \ldots, Y_q = -\emptyset$ dans $J(\mathcal{F} \setminus x) \times K_2$,

où $X_2 = Y_2 = \{y\}$. Nous pouvons donc construire le cycle hamiltonien suivant dans $J(\mathcal{F}) \times K_2$:

$+\emptyset = Y_1, X_1 \cup \{x\}, X_p \cup \{x\}, X_{p-1} \cup \{x\}, \ldots, X_2 \cup \{x\}, Y_2, Y_3, \ldots, Y_q = -\emptyset$. Ce cycle contient les arêtes $[+\emptyset, -\emptyset]$ et $[+\emptyset, +\{x\}]$. □

Voyons la construction à la Fig. 1.5

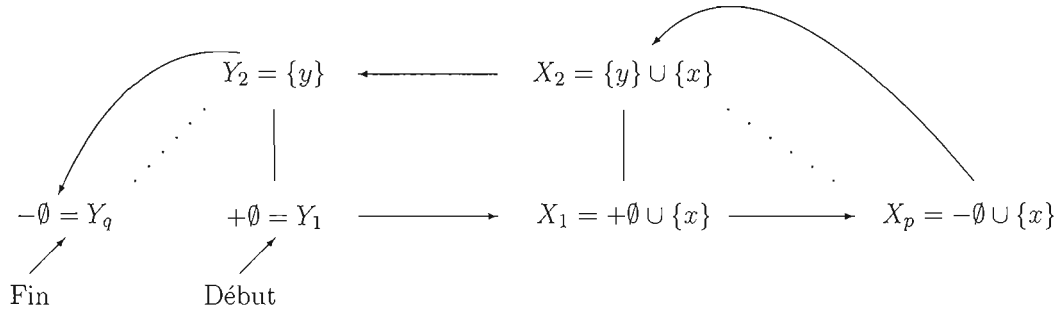

Fig. 1.5 Cycle dans $J(\mathcal{F}) \times K_2$ obtenu par composition des deux cycles dans $J(\mathcal{F}/x) \times K_2$ et $J(\mathcal{F} \setminus x) \times K_2$

Le chemin hamiltonien dans $J(\mathcal{F}) \times K_2$ a la propriété suivante : chaque idéal I (élément de $\mathcal{F}$) situé entre deux idéaux signés $\pm F$ et $\mp F$ vérifie $F \subset I$. Par conséquent, la liste d'éléments de $\mathcal{F}$ (ensembles réalisables) possède la structure de chaînes de parenthèses bien imbriquées avec $|\mathcal{F}|$ types de parenthèses.

Corollaire 1.1 *(Pruesse et Ruskey, 93) Pour n'importe quel antimatroïde $(E, \mathcal{F})$, il y a une liste $X_1, X_2, \ldots, X_{|\mathcal{F}|}$ d'ensembles de $\mathcal{F}$ telle que les ensembles successifs, y compris le premier et le dernier, diffèrent par deux éléments au plus.*

Preuve : Comme $J(\mathcal{F})$ est biparti, nous appliquons le lemme 1.1 pour conclure que $J(\mathcal{F})$ est hamiltonien. □

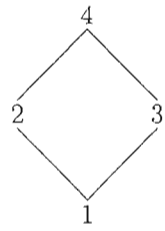

Fig. 1.6 Exemple d'un poset permettant de passer par tous les cas du théorème 1.1

1.4 Trace de l'algorithme par un exemple

Exemple 1.3 *Dans l'exemple de la Fig. 1.6 nous trouvons les deux cas traités dans la preuve du théorème 1.1.*

Soit $\mathcal{F} = \{\emptyset, 1, 12, 13, 123, 1234\}$.

Alors nous voyons que nous avons un seul singleton $\{1\}$. Ainsi, le premier cas de la preuve du théorème 1.1 s'applique. Il suffit alors d'ajouter $+\emptyset$ et $-\emptyset$ aux extrémités du chemin et 1 aux idéaux du chemin obtenu à partir de :

$\mathcal{F}' = \mathcal{F}/1 = \{\emptyset, 2, 3, 23, 234\}$.

Ici, nous voyons que nous avons les deux singletons $\{2\}$ et $\{3\}$, d'où selon le deuxième cas de la preuve du théorème 1.1, il suffit de composer les deux cycles (selon la Fig. 1.5) obtenus à partir de :

$\mathcal{F}'/2 = \{\emptyset, 3, 34\}$ *et* $\mathcal{F}' \setminus 2 = \{\emptyset, 3\}$.

Le premier satisfait le premier cas et le deuxième satisfait le cas où $|n| = 1$, d'où les graphes dans la Fig. 1.7. Selon la Fig. 1.5, nous pouvons composer ces deux posets, dont le graphe est à la Fig. 1.8. En l'écrivant d'une façon plus élégante, nous trouvons le graphe de $\mathcal{F}/1$ dans la Fig. 1.9. Finalement, le graphe de $\mathcal{F}$ est à la Fig. 1.10.

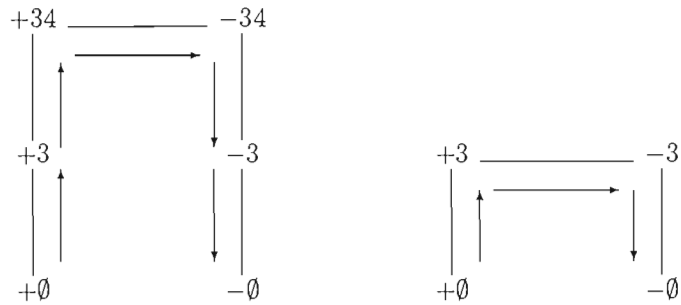

Fig. 1.7 Graphes de $\mathcal{F}'/2$ et de $\mathcal{F}' \setminus 2$.

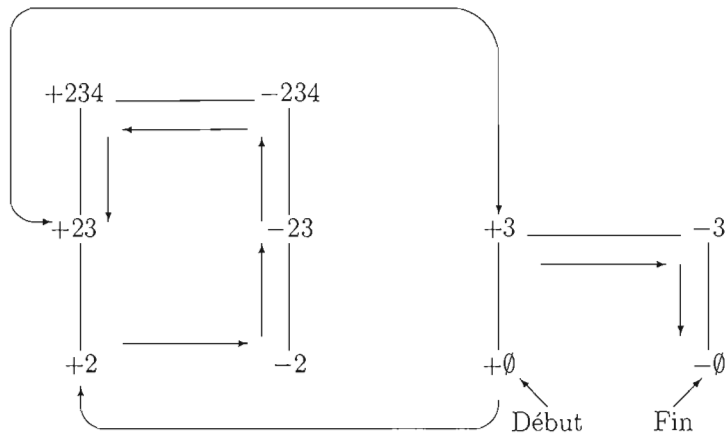

Fig. 1.8 Graphe de $\mathcal{F}/1$ qui est le composé des graphes de $\mathcal{F}'/2$ et de $\mathcal{F}' \setminus 2$.

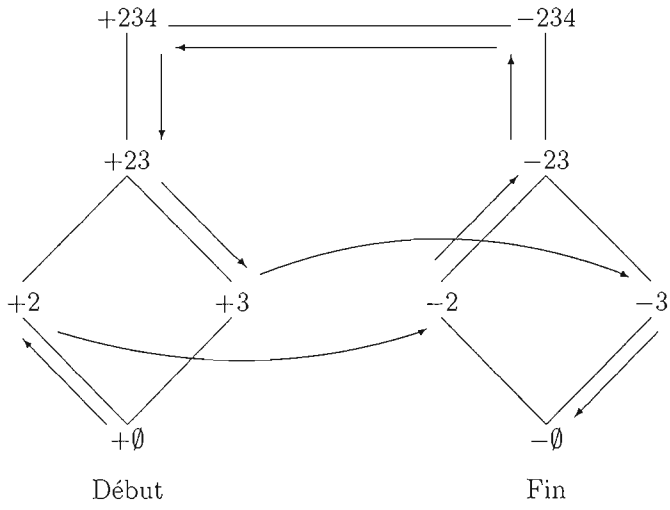

Fig. 1.9 Graphe de $\mathcal{F}/1$.

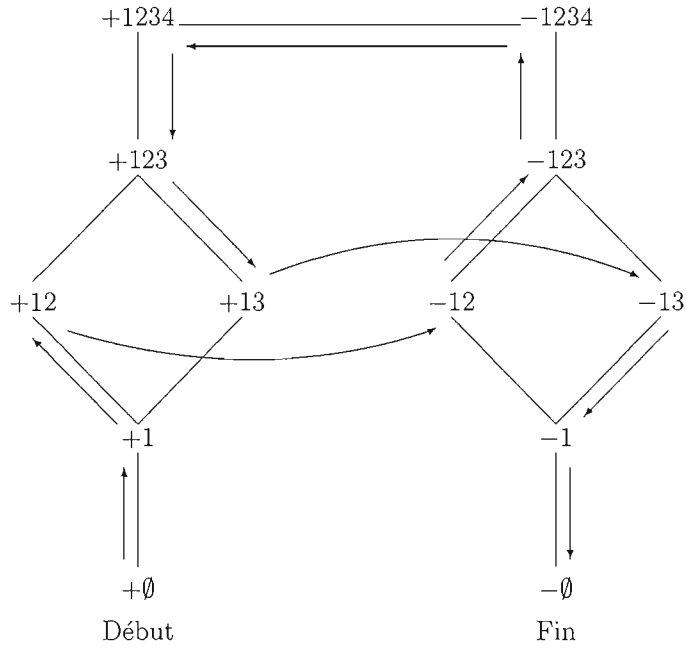

Fig. 1.10 Graphe de $\mathcal{F}$.

1.5 Programme

Ruskey a implanté le théorème 1.1 pour obtenir tous les idéaux d'un poset. Une démonstration se trouve sur sa page web :
www.theory.cs.uvic.ca/$\sim$ cos/gen/pose.html
Nous choisissons sur cette page, le type : *« Ideals in Gray code order »*, puis nous lui donnons n, le nombre d'éléments du poset, et l'ordre partiel (par exemple $\{1 < 2, 3 < 4\}$), enfin nous choisissons *« Generate »*.

1.6 Algorithme récursif de Ruskey

L'algorithme récursif de Ruskey est à la Fig. 1.11

1.6.1 Fonctions

Les fonctions utilisées par l'algorithme de la Fig. 1.11 ont les significations suivantes :

Update : Mise à jour de *num* et d'autres variables après avoir éliminé *min* de *poset*.

Recover : A l'effet inverse de *Update*, i.e., retrouve *num* et d'autres variables avant d'entrer dans l'appel récursif « Ideal ».

Print : Imprime l'idéal une fois sur deux.

1.6.2 Paramètres et variables locales

Les paramètres ont les significations suivantes :

poset : C'est l'ensemble partiellement ordonné représenté par un mot binaire. Ce mot sera positif et long pour qu'il puisse représenter un poset de taille assez grande.

dir : Selon le théorème 1.1, au début, la direction *dir* est FORWARD. Ensuite, si nous tombons dans le cas 1, *dir* ne change pas ; mais si nous tombons dans le cas 2, le cycle devient la composition des deux cycles, comme illustré dans la Fig. 1.5. Dans ce deuxième cas, *dir*=BACKWARD pour le *poset* privé de *min* et *dir*=FORWARD pour le *poset* privé de la branche qui contient *min*.

```
Procédure Ideal(long int non-négatif poset, Direction dir, int two_flag, int mask, int num)
    si (poset ≠ 0) alors
        si (2 minima et two_flag) alors {si cette itération possède 2 minima}
                                        {ainsi que la précédente}
            min ← Y
        sinon
            min ← le minimal de poset
        fin si
        si (num = 1) alors
            enlever min de poset
            num ← Update(poset, min, num)
            mettre min dans mask
            Print(mask)
            Ideal(poset, dir, 0, mask, num)
            Print(mask)
            num ← Recover(poset, min, num)
        sinon
            poset1 ← poset après la suppression de min
            Y ← le minimal de poset1
            res ← mask après l'ajout de min
            si (dir = FORWARD) alors
                Print(res)      {Dans Print, il y a un paramètre qui permet}
                Print(res)      {d'imprimer res une fois tous les deux appels}
                num ← Update(poset1, min, num)
                Ideal(poset1, BACKWARD, 1, res, num)
                num ← Recover(poset1, min, num)
                poset ← poset après la suppression de la branche qui contient min
                Y ← le minimal de poset
                Ideal(poset, FORWARD, 1, mask, num)
            sinon
                poset ← poset après la suppression de la branche qui contient min
                Ideal(poset, BACKWARD, 1, mask, num-1)
                Y ← le minimal de poset1
                num ← Update(poset1, min, num)
                Ideal(poset1, FORWARD, 1, res, num)
                num ← Recover(poset1, min, num)
                Print(res)
                Print(res)
            fin si
        fin si
    fin si
```

Fig. 1.11 Algorithme récursif de Ruskey.

two_flag : Au début *two_flag* est faux ; ensuite, si nous tombons dans le cas 1, *two_flag* ne change pas ; mais si nous tombons dans le cas 2, alors *two_flag* devient vrai, i.e., nous avons au moins deux minima : le premier est *min* et le deuxième est Y. Si nous tombons dans le cas 2 pour le prochain appel récursif, nous reprendrons Y comme *min*.

mask : Dans cette variable, nous mettons les *min* au fur et à mesure, afin d'obtenir un idéal à chaque ajout ; au début *mask* est initialisé à 0.

num : C'est le nombre de minima dans le *poset* courant ; ce paramètre nous indique si nous sommes dans le premier ou le deuxième cas du théorème 1.1.

Les variables locales ont les significations suivantes :

min : Le premier minimum du *poset* courant.

poset1 : Comme *poset*.

res : Comme *mask*.

Remarque 1.1 *La complexité temporelle de cet algorithme est* $O(n \cdot e(\mathcal{P}))$ *où* $e(\mathcal{P}) = |J(\mathcal{P})|$ *est le nombre d'idéaux du poset* $\mathcal{P}$.

1.7 Algorithme non-récursif

Au lieu de faire des appels récursifs, nous allons empiler les informations nécessaires pour ensuite les dépiler et faire des appels de telle sorte que l'algorithme devienne itératif. Les informations nécessaires empilées sont : **poset**, **min**, **num**, **mask** et **flag** ; les quatre premières ont les significations déjà données dans 1.6.2 et ici *flag* prend 4 valeurs qui sont relatives au théorème 1.1 :

1 : Pour dire que nous sommes dans le premier cas.

2 : Pour dire que nous sommes dans le deuxième cas avec *dir*=FORWARD.

3 : Pour dire que nous sommes dans le deuxième cas avec *dir*=BACKWARD et pour utiliser ces informations pour le prochain appel.

4 : Pour dire que nous sommes dans le deuxième cas avec *dir*=BACKWARD et pour utiliser ces informations pour imprimer l'idéal $Print(mask)$ et appeler *Recover*.

L'algorithme itératif est à la Fig. 1.12.

Nous voyons que la boucle *tant que* vérifie, pour chaque itération, si $poset \neq 0$. Si ce n'est pas le cas, elle vérifie si $pile \neq vide$. Sinon ($poset$=0 et $pile = vide$), elle s'arrête.

Nous avons ajouté une fonction à l'algorithme itératif qui permet de vérifier au début, à l'entrée de si (num=1), si le *poset* est un ordre total ; dans ce cas, les idéaux seront déterminés en temps constant amorti.

1.7.1 Comparaison des deux algorithmes

Nous avons testé les deux programmes qui implantent les algorithmes récursif et itératif, sur une vingtaine d'exemples de posets de petites tailles. Nous avons remarqué que dans tous les cas, l'algorithme itératif est plus rapide, mais seulement de 10 à 30 millisecondes (voir lc tab. 1.1).

1.7.2 Fonctionnement

Le poset est fourni sous la forme d'un fichier texte dans lequel figurent le nombre d'éléments du poset ainsi que les couples représentant la relation d'ordre partiel.

Dans ce cas, à partir d'une fenêtre DOS, il faut saisir le nom du fichier $C++$ exécutable (dont le code source est dans l'annexe A), suivi du nom de fichier texte qui représente le poset. À la sortie, nous obtenons :

- le nombre d'éléments du poset ;
- l'ensemble des couples représentant le poset ;
- les idéaux sous forme d'ensembles ;
- le nombre d'idéaux.

```
Procédure IdealIteratif(long int non-négatif poset, Direction dir, int two_flag, int mask, int num)
  tant que (pile non vide ou poset ≠ 0) faire
          si (poset ≠ 0) alors
            si (2 minima et two_flag) alors {si cette itération possède 2 minima}
                                            {ainsi que la précédente}
              min ← Y
            sinon
              min ← le minimal de poset
            fin si
            si (num = 1) alors
              enlever min de poset
              num ← Update(poset, min, num)
              mettre min dans mask
              Print(mask)
              two_flag ← 0
              empiler(min, num, poset, mask, flag=1)
            sinon
              poset1 ← poset après la suppression de min
              Y ← le minimal de poset1
              res ← mask après l'ajout de min
              si (dir = FORWARD) alors
                  Print(res)
                  Print(res)
                  num ← Update(poset1, min, num)
                  empiler(min, num, poset, mask, 2)
                  poset← poset1 ; dir← BACKWARD ; mask← res ; two_flag← 1 ;
              sinon
                  empiler(min, num, poset1, res, 4)
                  empiler(min, num, poset1, res, 3)
                  poset ← poset après la suppression de la branche qui contient min
                  two_flag← 1 ; num← num-1 ; dir← BACKWARD ;
              fin si
            fin si
          sinon
            dépiler(min, num, poset, mask, flag)
            cas flag de
                  1 : Print(mask)
                      num ← Recover(poset, min, num)
                  2 : poset1 ← poset après la suppression de min
                      num ← Recover(poset1, min, num)
                      poset ← poset après la suppression de la branche qui contient min
                      Y ← le minimal de poset ; dir← FORWARD ; two_flag← 1
                  3 : num ← Update(poset, min, num)
                      Y ← le minimal de poset ; dir← FORWARD ; two_flag← 1
                  4 : Print(mask)
                      Print(mask)
                      num ← Recover(poset, min, num)
          fin si
  fin {tant que}
```

Fig. 1.12 Algorithme itératif.

Poset	Idéal Rec	Idéal Iter	la différence de temps en millisecondes
$\{1<2, 2<3\}$	140	125	15
$\{1<2, 2<4, 3<4\}$	170	155	15
$\{1<2, 1<3, 2<4, 3<4\}$	165	155	10
$\{1<2, 3<4\}$	200	185	15
$\{1<2, 2<3, 3<4, 4<5\}$	160	150	10
$\{1<2, 1<3, 2<4, 3<5, 4<6, 5<6\}$	215	200	15
$\{1<2, 1<5, 2<3, 2<4, 3<6, 4<6, 5<6\}$	235	225	10
$\{1<2, 3<4, 5<6\}$	395	380	15
$\{1<2, 1<3, 2<4, 3<5, 4<6, 5<7, 6<8, 7<8\}$	290	280	10
$\{1<2, 2<3, \cdots, 7<8\}$	205	195	10
$\{1<2, 1<7, 2<3, 3<4, 3<5, 4<6, 6<9, 5<9,$ $7<8, 8<9\}$	380	355	25
$\{1<2, 2<3, \cdots, 8<9\}$	205	190	15
$\{1<2, 2<3, \cdots, 19<20\}$	425	415	10
$\{1<2, 2<3, \cdots, 29<30\}$	625	615	10
$\{1<3, 1<6, 2<4, 2<5, 4<7, 5<8, 7<9, 8<9\}$	675	645	30
$\{1<2, 2<3, 3<4, 4<5, 6<7, 7<8, 8<9, 9<10\}$	490	465	25

Tab. 1.1 La première colonne est le poset, la deuxième est le temps mis par l'algorithme récursif et la troisième est le temps mis par l'algorithme itératif.

1.8 Exemple

Supposons que le poset est donné sous la forme du fichier texte suivant :

6 1 2 1 5 2 3 2 4 3 6 4 6 5 6 0 0,

où le premier nombre indique le nombre d'éléments du poset et les deux zéros à la fin indiquent la fin du fichier texte, ce qui est équivalent au poset $\{ 1 < 2 , 1 < 5 , 2 < 3 , 2 < 4 , 3 < 6 , 4 < 6 , 5 < 6 \}$.

Après l'exécution du programme, nous avons la sortie suivante :

le nombre d'éléments du poset est : 6

le poset est :

$\{ 1 < 2 , 1 < 5 , 2 < 3 , 2 < 4 , 3 < 6 , 4 < 6 , 5 < 6 \}$.

Les idéaux en ordre code Gray sont :

{}

{1 2}

{1 2 4}

{1 2 3 4}

{1 2 3}

{1 2 3 5}

{1 2 3 4 5}

{1 2 3 4 5 6}

{1 2 4 5}

{1 2 5}

{1 5}

{1}

12 idéaux au total.

CHAPITRE II

CODE GRAY POUR LES IDÉAUX D'UN POSET FORÊT

Dans ce chapitre, nous allons montrer l'existence d'un code Gray pour un poset forêt, i.e., dont le diagramme de Hasse est une forêt.

2.1 Motivation

Le problème de génération des idéaux d'un poset arbre en ordre lexicographique a été considéré par Ruskey (Ruskey, 81) et motivé par le problème de partage des réseaux. Soit $G = (V, E)$, un graphe non-orienté pour lequel à chaque arête e, est associée une capacité $c(e)$ et à chaque sommet v, est associé un poids $w(v)$. Une *coupe* est une partition $\{X, V \setminus X\}$ de l'ensemble de sommets V en deux sous-ensembles disjoints et non-vides. La *capacité* d'une coupe $\{X, V \setminus X\}$ est la somme des capacités $c(e)$, où e est une arête avec une extrémité dans X et l'autre dans $V \setminus X$. Étant donnés une constante W et un sommet distingué v, considérons le problème de trouver une coupe de capacité minimale $\{X, V \setminus X\}$ telle que $v \in X$ et $\sum_{x \in X} w(x) \leq W$. Ce problème survient dans l'assignation des tâches dans un système distribué. Malheureusement, ce problème est *NP*-complet (Ruskey, 78).

Hu et Ruskey (Hu et Ruskey, 80) et Rao, Stone et Hu (Rao, Stone et Hu) ont abordé ce problème avec une approche énumérative. Soit T, l'arbre de coupe pour le réseau et supposons que T possède v comme racine. Dans ces deux articles, les auteurs ont montré que la solution optimale pour le problème de partage doit être une coupe $(X, V \setminus X)$

pour laquelle l'ensemble des sommets dans X est connexe dans T ; autrement dit, X doit être un idéal de T. Donc, en générant tous les idéaux de T, la solution optimale peut être obtenue. Dans ce sens, il est clair que nous allons avoir des avantages calculatoires dans le code Gray, puisqu'il est très simple de modifier une coupe si un seul sommet change.

Koda et Ruskey croient que les posets fournissent un cadre pour travailler avec les codes Gray combinatoires (Koda et Ruskey, 93). Ce point de vue pour les extensions linéaires de posets a été exploré, par exemple dans (Ruskey, 92).

Deux célèbres codes Gray sont le *Binary Reflected Gray Code* (BRGC) pour la génération de sous-ensembles (Bitner, Ehrlich et Reingold, 76) et l'algorithme de Johnson-Trotter pour la génération de permutations (Johnson, 63; Trotter, 62). Les résultats de (Ruskey, 92) fournissent une généralisation naturelle de l'algorithme de Johnson-Trotter et les résultats de (Koda et Ruskey, 93) fournissent une généralisation naturelle des résultats connus de BRGC.

Dans ce chapitre, nous allons présenter l'algorithme de (Koda et Ruskey, 93) pour la génération du code Gray pour les idéaux d'un poset forêt ; cet algorithme occupe un espace de $O(n)$, où n est le nombre d'éléments du poset. Pour chaque itération, cet algorithme fait une fouille partielle pour l'idéal courant et exige un temps de $O(nN)$, où N est le nombre d'idéaux du poset (Koda et Ruskey, 93). Un autre algorithme peut être trouvé dans la même référence. Ce dernier évite la fouille, est sans boucle (voir la définition 3.7) et exige un temps de $O(N)$.

2.2 Définitions et notations

Soit $\mathcal{P}$ un poset, $R(\mathcal{P})$ est la relation d'ordre partielle et $S(\mathcal{P})$ est l'ensemble de base.

Définition 2.1 *Nous disons que a et b sont* ***comparables*** *si $a \preceq b$ ou $b \preceq a$, sinon nous disons qu'ils sont* ***incomparables*** *et nous écrivons $a \parallel b$.*

Définition 2.2 *Une* ***antichaîne*** *est un sous-ensemble de $S(\mathcal{P})$ dans lequel toute paire d'éléments est incomparable. Une* ***chaîne*** *est un sous-ensemble de $S(\mathcal{P})$ dans lequel toute paire d'éléments est comparable.*

Définition 2.3 *Nous disons que b* ***couvre*** *a si $a \prec b$ et s'il n'existe aucun c dans $S(\mathcal{P})$ tel que $a \prec c \prec b$. Si v couvre un élément w, alors v est un* ***enfant*** *de w et w est le* ***parent*** *de v.*

Notation 2.1 *Notons* ***Enfant****(v), l'ensemble des enfants de v,* ***parent****(v), le parent de v s'il existe et* ***Frère****(v), l'ensemble de frères de v y compris v.*

Définition 2.4 *Un* ***poset forêt*** *$\mathcal{F}$ est celui dans lequel chaque élément couvre au plus un autre élément.*

Définition 2.5 *Un élément a est dit* ***minimal*** *dans $\mathcal{P}$ s'il n'existe aucun élément b tel que $b \prec a$.*

Définition 2.6 *Un* ***poset arbre*** *est un poset forêt avec exactement un élément minimal.*

Notation 2.2 *Notons* ***Racine****$(\mathcal{F})$, l'ensemble des racines qui sont les éléments minimaux de la forêt, et* ***Feuille****$(\mathcal{F})$, l'ensemble des feuilles qui sont les éléments maximaux de la forêt.*

Définition 2.7 *Supposons que a et b sont deux éléments incomparables de $S(\mathcal{P})$, tels que a a la même relation que b avec tous les autres éléments de $S(\mathcal{P})$; plus précisement, supposons que pour tout $c \in S(\mathcal{P})$, $c \prec a$ si et seulement si $c \prec b$, et $a \prec c$ si et seulement si $b \prec c$. Alors nous appelons a et b deux* ***frères*** *« **Sibling** ».*

Remarque 2.1 *Si parent(v)=w, alors* ***Frère****(v)=Enfant(w) ; si v n'a pas de parent, alors* ***Frère****(v)= Racine$(\mathcal{F})$.*

Définition 2.8 *Nous disons que deux posets $\mathcal{P}$ et $\mathcal{Q}$ sont* ***disjoints*** *si $S(\mathcal{P})$ et $S(\mathcal{Q})$ sont disjoints.*

Il y a deux manières habituelles pour combiner deux posets et en obtenir un nouveau.

Définition 2.9 *La* ***somme directe*** *de deux posets disjoints $\mathcal{P}$ et $\mathcal{Q}$ est le poset $\mathcal{P}+\mathcal{Q}$ sur l'union $S(\mathcal{P}) \cup S(\mathcal{Q})$ tel que $x \preceq y$ dans $\mathcal{P}+\mathcal{Q}$ si*
- *soit $x, y \in \mathcal{P}$ et $x \preceq y$ dans $\mathcal{P}$,*
- *soit $x, y \in \mathcal{Q}$ et $x \preceq y$ dans $\mathcal{Q}$.*

Définition 2.10 *La* ***somme ordonnée*** *de deux posets disjoints $\mathcal{P}$ et $\mathcal{Q}$ est le poset $\mathcal{P} \oplus \mathcal{Q}$ sur l'union $S(\mathcal{P}) \cup S(\mathcal{Q})$ tel que $x \preceq y$ dans $\mathcal{P} \oplus \mathcal{Q}$ si*
- *soit $x \preceq y$ dans $\mathcal{P}+\mathcal{Q}$,*
- *soit $x \in \mathcal{P}$ et $y \in \mathcal{Q}$.*

Remarque 2.2 *Notons qu'un poset forêt peut être construit en appliquant récursivement les opérations de la somme directe et ordonnée, comme c'est le cas d'un poset série-parallèle, en commençant par les posets possédant un seul élément.*

Définition 2.11 *Soient A et B deux ensembles distincts d'entiers naturels. Nous disons que A est* ***lexicographiquement plus petit*** *que B si le plus grand élément de $(A \cup B) \setminus (A \cap B)$ est dans B.*

Exemple 2.1 *$\{2,5,6\}$ est lexicographiquement plus petit que $\{1,3,5,6\}$.*

2.3 Distance $d(\mathcal{P})$ d'un poset forêt

L'ensemble des idéaux d'un poset $\mathcal{P}$, dont la relation d'ordre est l'inclusion des ensembles, forme un ***treillis distributif*** $J(\mathcal{P})$. Nous savons que pour chaque treillis distributif L, il existe un et un seul poset $\mathcal{P}$ pour lequel $J(\mathcal{P})$ est isomorphe à L. Notons que

$J(\mathcal{P})$ est ***connexe*** et ***biparti***. Il est connexe puisque chaque treillis distributif possède un élément maximal et un élément minimal. La bipartition des sommets de $J(\mathcal{P})$ est en deux ensembles $\mathbf{Pair}(\mathcal{P})$ et $\mathbf{Impair}(\mathcal{P})$, selon la parité du nombre d'éléments de l'idéal. Dans cette section, nous montrons qu'il existe un chemin hamiltonien dans $J(\mathcal{P})$ si $\mathcal{P}$ est un poset forêt. Une condition nécessaire pour l'existence d'un chemin hamiltonien dans $J(\mathcal{P})$ est que $|d(\mathcal{P})| \leq 1$. Pourtant, cette condition n'est pas suffisante tel qu'indiqué dans la Fig. 1.3.

Définition 2.12 *La **différence de parité** $d(\mathcal{P})$ est définie par* $|\ Pair(\mathcal{P})\ |\ -\ |\ Impair(\mathcal{P})\ |$.

Notation 2.3 *Par abus de notation, $J(\mathcal{P})$ désigne le **diagramme de Hasse**, considéré comme un graphe non-orienté du treillis des idéaux de $\mathcal{P}$. Dans ce contexte, nous noterons $J(\mathcal{P})$, le **graphe idéal** de $\mathcal{P}$.*

Notation 2.4 *Notons $E(\mathcal{P})$, l'ensemble des extensions linéaires de $\mathcal{P}$ et $e(\mathcal{P})$, sa taille $|E(\mathcal{P})|$.*

Lemme 2.1

$$e(\mathcal{P} + \mathcal{Q}) = e(\mathcal{P})e(\mathcal{Q}).$$

Preuve : D'après la définition de la somme directe, un idéal dans $\mathcal{P} + \mathcal{Q}$ est l'union d'un idéal dans $\mathcal{P}$ avec un idéal dans $\mathcal{Q}$, d'où l'égalité ci-dessus. □

Lemme 2.2

$$e(\mathcal{P} \oplus \mathcal{Q}) = -1 + e(\mathcal{P}) + e(\mathcal{Q}).$$

Preuve : Selon la définition $\mathcal{P} \oplus \mathcal{Q} = J(\mathcal{P}) \cup \{S(\mathcal{P}) \cup E : E \in S(\mathcal{Q})\}$, et en remarquant que lorsque nous ajoutons $S(\mathcal{P})$ à chaque élément E de $S(\mathcal{Q})$, nous aurons deux fois $S(\mathcal{P})$: une dans $J(\mathcal{P})$ et l'autre $S(\mathcal{P}) \cup \emptyset$, qui est dans $\mathcal{P} \oplus \mathcal{Q}$. Autrement dit, un idéal de $\mathcal{P} \oplus \mathcal{Q}$ est soit un idéal de $\mathcal{P}$, soit l'union de $S(\mathcal{P})$ avec un idéal non-vide de $\mathcal{Q}$, d'où l'égalité ci-dessus. □

De façon similaire, nous avons :

Lemme 2.3

$$d(\mathcal{P} + \mathcal{Q}) = d(\mathcal{P})d(\mathcal{Q}).$$

Preuve : Pour chaque idéal I de $\mathcal{P}$:

- si $|I|$ est pair, alors en associant I à chaque idéal J de Q, nous aurons : $|I \cup J|$ pair si et seulement si $|J|$ pair, i.e., $|I \cup J|$ et $|J|$ ont la même parité.
- si $|I|$ est impair, alors en associant I à chaque idéal J de Q, nous aurons : $|I \cup J|$ impair si et seulement si $|J|$ pair, i.e., $|I \cup J|$ et $|J|$ ont des parités différentes.

Donc,

$$Pair(\mathcal{P} + \mathcal{Q}) = Pair(\mathcal{P})Pair(\mathcal{Q}) + Impair(\mathcal{P})Impair(\mathcal{Q})$$

et

$$Impair(\mathcal{P} + \mathcal{Q}) = Pair(\mathcal{P})Impair(\mathcal{Q}) + Impair(\mathcal{P})Pair(\mathcal{Q}),$$

d'où l'égalité ci-dessus. □

Lemme 2.4

$$d(\mathcal{P} \oplus \mathcal{Q}) = \begin{cases} -1 + d(\mathcal{P}) + d(\mathcal{Q}) & si\ |\mathcal{P}|\ est\ pair \\ +1 + d(\mathcal{P}) - d(\mathcal{Q}) & si\ |\mathcal{P}|\ est\ impair. \end{cases}$$

Preuve : Si $|\mathcal{P}|$ est pair, alors selon la définition 2.10 de $\mathcal{P} \oplus \mathcal{Q}$, en ajoutant un nombre pair d'éléments à chaque idéal de $J(\mathcal{Q})$, $d(\mathcal{Q})$ ne va pas changer ; par contre $S(\mathcal{P})$ sera compté deux fois, une dans $d(\mathcal{P})$ et une dans $d(\mathcal{P} \oplus \mathcal{Q})$ (cette deuxième fois correspond à $S(\mathcal{P}) \cup \emptyset$), d'où nous ajoutons $(d(\mathcal{Q}) - 1)$ à $d(\mathcal{P})$. Regardons l'exemple 2.2.
Si $|\mathcal{P}|$ est impair, alors selon la définition de $\mathcal{P} \oplus \mathcal{Q}$, en ajoutant $S(\mathcal{P})$ à chaque élément de $S(\mathcal{Q})$, les idéaux dans $J(\mathcal{Q})$ changent de cardinalité de paire à impaire et vice versa, d'où nous ajoutons $-(d(\mathcal{Q}) - 1)$ à $d(\mathcal{P})$. Regardons l'exemple 2.3. □

Exemple 2.2 *Étant donné $\mathcal{P}$ et $\mathcal{Q}$ de la Fig. 2.1, nous trouvons $\mathcal{P} \oplus \mathcal{Q}$ d'après la définition de la somme ordonnée. Puis dans la Fig. 2.2, nous trouvons leur distance correspondante.*

Exemple 2.3 *Étant donné $\mathcal{P}$ et $\mathcal{Q}$ de la Fig. 2.3, nous trouvons $\mathcal{P} \oplus \mathcal{Q}$ d'après la définition de la somme ordonnée. Puis dans la Fig. 2.4, nous trouvons leur distance correspondante.*

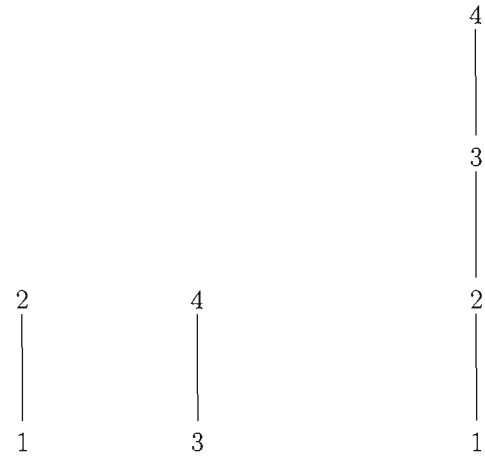

Fig. 2.1 En ordre, $\mathcal{P}$, $\mathcal{Q}$ et $\mathcal{P} \oplus \mathcal{Q}$.

2.4 Théorème constructif pour un code Gray

Dans cette section, nous allons présenter la preuve de l'existence d'un chemin hamiltonien dans $J(\mathcal{P})$ si $\mathcal{P}$ est un poset forêt (Koda et Ruskey, 93).

Lemme 2.5 *Soient $\mathcal{P}$ et $\mathcal{Q}$, deux posets non-vides dont le graphe des idéaux de chacun possède un chemin hamiltonien commençant à $\emptyset$. Alors $J(\mathcal{P} + \mathcal{Q})$ possède un chemin hamiltonien commençant à $\emptyset$. De plus, si $|J(\mathcal{P})|$ (ou $|J(\mathcal{Q})|$) est pair, alors $J(\mathcal{P} + \mathcal{Q})$ possède un cycle hamiltonien.*

Lemme 2.6 *Soit $\mathcal{P}$ un poset pour lequel $J(\mathcal{P})$ possède un chemin hamiltonien commençant à $\emptyset$ et terminant à $S(\mathcal{P})$. Soit $\mathcal{Q}$ un poset pour lequel $J(\mathcal{Q})$ possède un chemin hamiltonien commençant à $\emptyset$. Alors il existe un chemin hamiltonien dans $J(\mathcal{P} \oplus \mathcal{Q})$ commençant à $\emptyset$.*

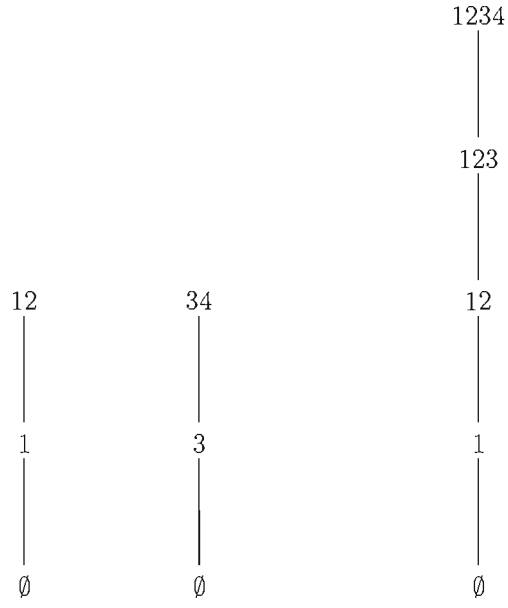

Fig. 2.2 En ordre, $J(\mathcal{P})$, $J(\mathcal{Q})$ et $J(\mathcal{P} \oplus \mathcal{Q})$, d'où $d(\mathcal{P})$=1, $d(\mathcal{Q})$=1 et $d(\mathcal{P} \oplus \mathcal{Q})$=1 .

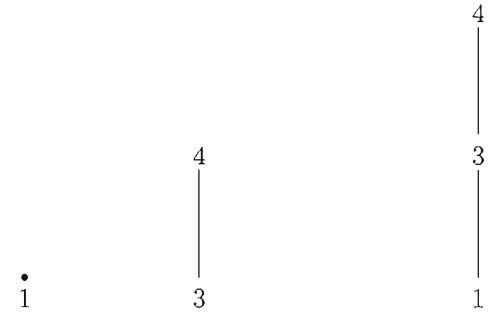

Fig. 2.3 En ordre, $\mathcal{P}$, $\mathcal{Q}$ et $\mathcal{P} \oplus \mathcal{Q}$.

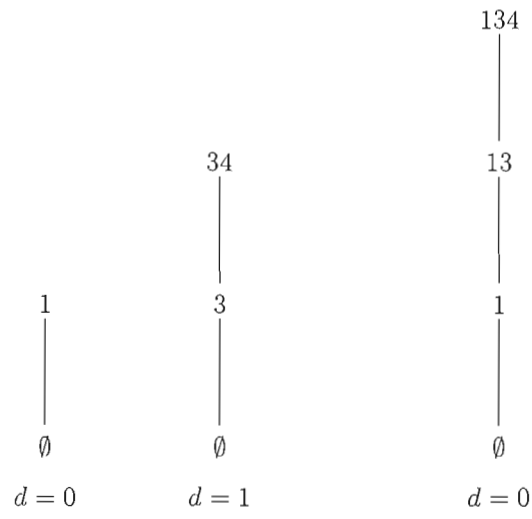

Fig. 2.4 En ordre, $J(\mathcal{P})$, $J(\mathcal{Q})$ et $J(\mathcal{P} \oplus \mathcal{Q})$, d'où $d(\mathcal{P})$=0, $d(\mathcal{Q})$=1 et $d(\mathcal{P} \oplus \mathcal{Q})$=0.

Le théorème suivant est une conséquence des lemmes 2.5 et 2.6, et de la remarque 2.2. Il peut être demontré par récurrence sur le nombre de noeuds dans la forêt.

Théorème 2.1 *Si $\mathcal{P}$ est un poset forêt, alors il existe un chemin hamiltonien dans $J(\mathcal{P})$.*

2.5 Algorithme

Le théorème 2.1 est constructif, mais il ne décrit pas un chemin hamiltonien particulier dans le cas de la somme directe. Dans cette section, nous allons décrire un algorithme qui trouve un tel chemin systématiquement (Koda et Ruskey, 93). Ce dernier article dit que cet algorithme est inspiré de la bijection entre le calcul binaire et le BRGC (Reingold, Nievergelt et Deo, 77; Wilf, 89).

Remarque. Dans cette section, $\mathcal{F}$ est un poset forêt avec n éléments, St est un ensemble de noeuds. Un noeud $x \in$ St est dit ***fixe***, sinon il est dit ***libre***.

Définition 2.13 *Soit F un idéal d'un poset forêt $\mathcal{F}$, et $v \notin St$.*
- Si $v \notin F$, et si v est une racine ou parent$(v) \in F$, alors v est ajoutable.
- Si $v \in F$, et si aucun enfant de v n'est dans F, alors v est supprimable.

Autrement dit, un noeud *libre* v est ajoutable si et seulement si $v \in Racine(\mathcal{F} \setminus F)$, et est supprimable si et seulement si $v \in Feuille(F)$. Un noeud qui est ajoutable ou supprimable est dit *changeable*. Étant donnés F et St, s'il existe un noeud *changeable* dans $\mathcal{F}$, alors $\mathcal{F}$ est *changeable* ; sinon $\mathcal{F}$ est inchangeable.

Remarque 2.3 *L'ensemble* St *croît lexicographiquement à chaque itération de l'algorithme P. Il suffit de remarquer qu'à la ligne 7 de l'algorithme P de la Fig. 2.5,* St *change d'une itération à la suivante en ajoutant un élément v et en enlevant les éléments inférieurs à v. Donc, St va toujours avoir un élément supérieur à ceux enlevés. Par conséquent, selon la définition 2.11* St *croît lexicographiquement.*

2.5.1 Fonctionnement de l'algorithme

Nous supposons que la forêt est étiquetée en ordre préfixe, et nous utilisons ces étiquettes lorsque nous consultons les noeuds de $\mathcal{F}$. Cet ordre préfixe peut être obtenu en étiquetant récursivement une racine, puis ses sous-arbres ; c'est la même procédure que la recherche en profondeur. Nous supposons que les étiquettes croissent de gauche à droite et que les étiquettes sont $1, 2, \ldots, n$.

L'algorithme fait une fouille de la forêt en ordre préfixe, changeant les noeuds de *fixe* à *libre* jusqu'à ce que nous trouvons le premier noeud *changeable* ; ce noeud est alors ajouté ou retiré de l'arbre selon qu'il est ajoutable ou supprimable. Nous voyons que la complexité de l'algorithme P de la Fig. 2.5 est $O(nN)$, où $N = e(\mathcal{F})$; ceci découle du fait que trouver *min* (la ligne 6 de l'algorithme P de la Fig. 2.5) coûte n et qu'il y a N idéaux.

1. $F \leftarrow \emptyset$;
2. $\mathrm{St} \leftarrow \emptyset$;
3. Sortir(F) ;
4. **tant que** $\mathcal{F}$ est changeable **faire**
5. **début**
6. $v \leftarrow \min\{v \in \mathcal{F} \mid v \text{ est changeable } \}$;
7. $\mathrm{St} \leftarrow \mathrm{St} \cup \{v\} \setminus \{w \in \mathcal{F} \mid w < v\}$;
8. **si** $v \in F$ **alors** $F \leftarrow F \setminus \{v\}$ **sinon** $F \leftarrow F \cup \{v\}$;
9. Sortir(F) ;
10. **fin** ;

Fig. 2.5 Algorithme P pour trouver un chemin hamiltonien dans $J(\mathcal{F})$ qui commence par $\emptyset$.

2.6 Trace de l'algorithme

Exemple 2.4 *La Fig. 2.6 illustre graphiquement la trace de l'algorithme P pour le graphe étiqueté qui se trouve en haut de cette même figure, et le Tab. 2.1 est la trace du même graphe.*

Remarque. *Le minimum (la ligne 6 de l'algorithme P de la Fig. 2.5) est pris selon le préordre des étiquettes.*
L'algorithme s'arrête lorsqu'il n'y a plus de noeud changeable ; dans ce cas, toutes les feuilles sont dans St.

v	St	F
	$\emptyset$	$\emptyset$
1	1	1
2	2	12
3	3	123
4	4	1234
3	34	124
5	5	1245
3	35	12345
4	45	1235
3	345	125
2	2345	15
6	6	156
2	26	1256
3	36	12356
4	46	123456
3	346	12456

Tab. 2.1 Trace de l'algorithme P de la Fig. 2.5 pour le graphe étiqueté de la Fig. 2.6.

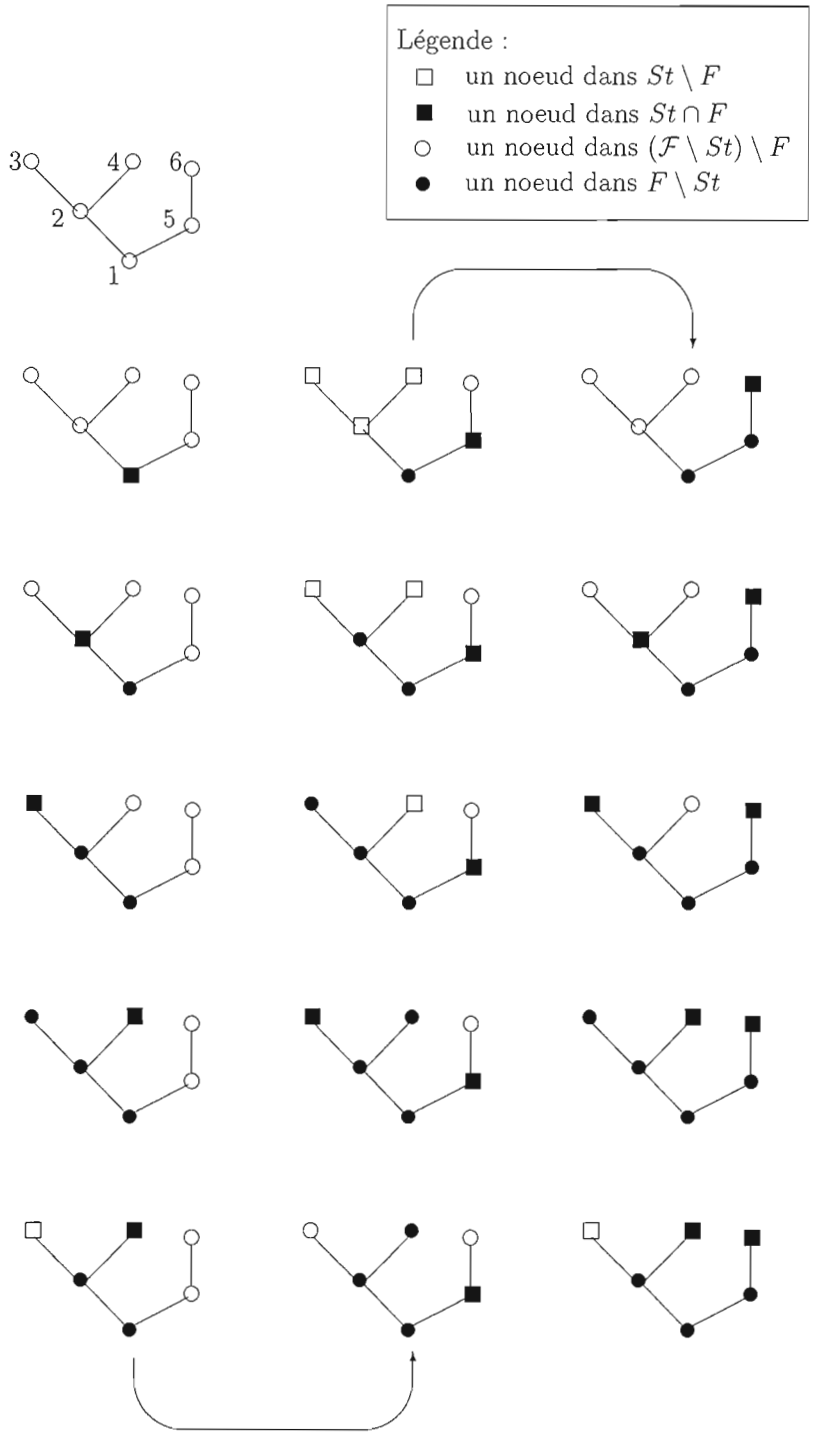

Fig. 2.6 Trace de l'algorithme P de la Fig. 2.5 pour le poset du graphe étiqueté ci-haut. Ordre à suivre : première et troisième colonnes, de haut en bas ; deuxième colonne, de bas en haut.

CHAPITRE III

ÉNUMÉRATION DES IDÉAUX D'UN POSET

Dans ce chapitre, nous ne présentons pas un code Gray, mais plutôt un algorithme pour énumérer les idéaux d'un poset en un temps amorti de $O(\log n)$ par idéal, alors que les algorithmes qui existent requièrent un temps amorti de $O(n)$ par idéal, où n est le nombre d'éléments du poset.

3.1 Définitions et notation

Définition 3.1 *Un* ***up-set*** *d'un poset $\mathcal{P}$ est un sous-ensemble U de $S(\mathcal{P})$ tel que $x \in U$ et $y \geq x$ implique que $y \in U$.*

Définition 3.2 *Un* ***down-set*** *d'un poset $\mathcal{P}$ est un sous-ensemble D de $S(\mathcal{P})$ tel que $x \in D$ et $y \leq x$ implique que $y \in U$.*

Remarque 3.1 *Les up-sets et down-sets sont appelés* ***idéaux****. Dans les chapitres précédents, nous utilisions les down-sets que nous appelions les idéaux.*

Dans ce chapitre, nous allons présenter un algorithme qui génère les up-sets d'un poset, et avec des modifications simples, cet algorithme génère les down-sets.

Notation 3.1 *Notons $\mathcal{U}(\mathcal{P})$, l'ensemble des up-sets du poset $\mathcal{P}$, et soit $u(\mathcal{P}) = |\mathcal{U}(\mathcal{P})|$.*

3.2 Comparaison avec la littérature

Schrage et Baker (Schrage et Baker, 78), Lawler (Lawler, 79), et Ball et Provan (Ball et Provan, 83) ont donné des algorithmes qui généraient $\mathcal{U}(\mathcal{P})$ en $O(n^2 \cdot u(\mathcal{P}))$, où $n = |S(\mathcal{P})|$. Steiner (Steiner, 86) était le premier à présenter un algorithme qui prenait un temps linéaire amorti $O(n \cdot u(\mathcal{P}))$. Pruesse et Ruskey (Pruesse et Ruskey, 93), comme nous avons déjà vu dans le chapitre 1, ont trouvé un algorithme qui génère les down-sets en ordre code Gray avec un temps linéaire amorti. Si le diagramme de Hasse du poset $\mathcal{P}$ est acyclique, alors l'algorithme de Beyer et Ruskey (Beyer et Ruskey, 89) et celui de Koda et Ruskey (Koda et Ruskey, 93) déjà vu dans le chapitre 2, peuvent générer les idéaux en temps constant amorti. Cependant, pour un poset arbitraire, le meilleur algorithme connu peut toujours prendre un temps de $O(n \cdot u(\mathcal{P}))$.
Nous allons présenter un algorithme qui génère $\mathcal{U}(\mathcal{P})$ en temps de $O(\log n \cdot u(\mathcal{P}))$ (Squire).

3.3 Récurrence

Squire a d'abord trouvé une récurrence pour les up-sets; ensuite, il a développé un algorithme basé sur cette récurrence (Squire).

Notation 3.2 *Soit $\mathcal{P} = (X, R)$, un poset où X est l'ensemble de base, et R est la relation rélflexive, antisymétrique et transitive sur X, et $x \in X$. Notons $U[x]$, l'ensemble up-set de x qui consiste des éléments $y \in X$ tels que $y \geq x$ et $D[x]$, l'ensemble down-set de x qui consiste des éléments $y \in X$ tels que $y \leq x$.*

Définition 3.3 *Soit U, un up-set de $\mathcal{P}$. Un sous ensemble Y de $X \setminus U$ est dit* **compatible** *avec U si $U \cup Y$ est aussi un up-set de $\mathcal{P}$.*

Notation 3.3 *Soit Y un sous-ensemble de $X \setminus U$ qui est compatible avec U. Notons $\mathcal{U}(\mathcal{P}, U, Y)$, l'ensemble des up-sets V de $\mathcal{P}$ tel que $U \subseteq V \subseteq (U \cup Y)$.*

Observation. L'ensemble $\mathcal{U}(\mathcal{P}, U, Y)$ contient tous les super-up-sets de U, i.e. contient tous les ensembles qui contiennent les up-sets de U, dont les éléments supplémentaires sont choisis dans Y.
Remarquons que $\mathcal{U}(\mathcal{P}) = \mathcal{U}(\mathcal{P}, \emptyset, Y)$.

Soit U, un up-set de $\mathcal{P}$ et $Y \subseteq X \setminus U$, un ensemble compatible avec U. Soit $V \in \mathcal{U}(\mathcal{P}, U, Y)$. Pour n'importe quel $x \in Y$, soit $x \in V$, soit $x \notin V$. Supposons que $x \in Y \cap V$. Dans ce cas, si $y \geq x$, alors $y \in V$, d'où $V \in \mathcal{U}(\mathcal{P}, U \cup U[x], Y \setminus U[x])$. Supposons maintenant que $x \in Y \setminus V$. Alors $y \leq x$ implique que $y \notin V$ (car si $y \in V$ et $x \geq y$, alors $x \in V$). Dans ce cas, $V \in \mathcal{U}(\mathcal{P}, U, Y \setminus D[x])$. Par conséquent, pour tout $x \in Y$ nous avons :

$$\mathcal{U}(\mathcal{P}, U, Y) = \mathcal{U}(\mathcal{P}, U \cup U[x], Y \setminus U[x]) \cup \mathcal{U}(\mathcal{P}, U, Y \setminus D[x]). \tag{3.1}$$

Observation. Dans l'équation (3.1), nous pouvons considérer U comme l'ensemble des éléments qui peuvent être dans le up-set et $X \setminus (U \cup Y)$ comme l'ensemble des éléments qui sont définitivement hors du up-set. Nous avons donc clairement $\mathcal{U}(\mathcal{P}, U, \emptyset) = \{U\}$, et l'équation fournit une décomposition récursive de $\mathcal{U}(\mathcal{P})$.

Définition 3.4 *Une* ***extension*** *de $\mathcal{P}$ est un poset $\mathcal{Q}$ tel que $S(\mathcal{P}) = S(\mathcal{Q})$ et $R(\mathcal{P}) \subseteq R(\mathcal{Q})$. Une extension de $\mathcal{P}$ qui est un ordre total est appelée une* ***extension linéaire*** *de $\mathcal{P}$.*

Nous nous intéressons aux algorithmes efficaces pour générer des objets combinatoires. Supposons que les objets sont representés par des suites de n éléments et que le nombre total d'objets est N.

Définition 3.5 *Nous disons que l'algorithme fonctionne en* ***temps constant amorti*** *si le temps total de calcul, excluant celui de la sortie, pour générer ces objets est $O(N)$. À une constante près, il n'y a pas d'algorithme plus rapide.*

Définition 3.6 *Si le temps total de calcul est $O(nN)$, alors nous disons que l'algorithme fonctionne en* ***temps linéaire amorti.***

Définition 3.7 *Un algorithme de génération est dit* ***sans boucle*** *si le temps de calcul entre les objets successifs est $O(1)$ dans le pire cas.*

3.4 Description de l'algorithme

Nous allons décrire l'algorithme présenté dans (Squire), basé sur l'équation (3.1). Soit $\mathcal{P} = (X, R)$, un poset, où $X = \{x_1, x_2, \cdots, x_n\}$. Le poset $\mathcal{P}$ est représenté par une matrice P de dimensions $n \times n$. Pour $1 \leq i, j \leq n$, $P[i, j] = 1$ si $x_i \leq x_j$ et $P[i, j] = 0$ sinon. Nous supposons que la matrice est globale (accessible globalement). Le up-set U est représenté par un tableau de longueur n tel que pour $1 \leq i \leq n$ et $x_i \in X \setminus Y$, $U[i] = 1$ si $x_i \in U$ et $U[i] = 0$ sinon. Aucune instruction n'est posée concernant la valeur de $U[i]$ quand $x_i \in Y$. Le sous-ensemble Y de $X \setminus U$ est représenté par une liste. Les éléments sont insérés dans cette liste dans un ordre cohérent avec R. La liste peut être vue comme une sous-suite d'une extension linéaire de $\mathcal{P}$. Notons ϵ la liste vide. La procédure récursive GenUpSets de la Fig. 3.1 génère $\mathcal{U}(\mathcal{P}, U, Y)$ en utilisant la décomposition de l'équation (3.1). La procédure RestoreU(U) restitue le tableau U aux valeurs qu'il avait au moment de l'appel de la procédure. Nous omettons les détails de RestoreU.

Remarque 3.2 *Pour trouver $\mathcal{U}(\mathcal{P})$, nous prenons U comme l'ensemble vide et Y est représenté par une liste (sous-suite d'une extension linéaire de $\mathcal{P}$).*

3.4.1 Clarification de l'algorithme

L'équation (3.1) permet à n'importe quel $x \in Y$ d'être utilisé dans la récurrence. Dans la ligne 5 de la Fig. 3.1, *l'élément au milieu* de Y est retiré pour être l'élément x utilisé dans la récurrence. L'algorithme aurait généré correctement $\mathcal{U}(\mathcal{P}, U, Y)$ si n'importe

```
1. procédure GenUpSets(tableau U, liste Y);
2. début
3.    si (Y = ϵ) alors retourner J;
4.    sinon
      début
5.       x_m ← l'élément au milieu de la liste Y;
6.       (* Générer U(P, U, Y \ D[x]) *)
7.       Y' ← ϵ;
8.       pour chaque x_i dans Y faire          (* éliminer x_m de l'idéal *)
9.          si (P[i, m] = 1) alors U[i] ← 0;
10.         sinon ajouter x_i à la liste Y';
11.      GenUpSets(U, Y');
12.      RestoreU(U);
13.      (* Générer U(P, U ∪ U[x], Y \ U[x]) *)
14.      Y' ← ϵ;
15.      pour chaque x_j dans Y faire          (* inclure x_m dans l'ideal *)
16.         si (P[m, j] = 1) alors U[j] ← 1;
17.         sinon ajouter x_j à la liste Y';
18.      GenUpSets(U, Y');
19.   fin;
20. fin.
```

Fig. 3.1 Algorithme pour générer les idéaux

quel $x \in Y$ était choisi dans la ligne 5. Or, si un x arbitraire était choisi dans la ligne 5, l'algorithme aurait pu dégénérer vers $O(n)$ par up-set. Alors le choix de *l'élément au milieu* est nécessaire pour assurer un temps de $O(\log n \cdot u(\mathcal{P}))$. Soit L, une extension linéaire de $\mathcal{P}$ insérée dans une liste. L'appel GenUpSets(U, L) génère tous les up-sets de $\mathcal{P}$ (le tableau U ne requiert pas d'initialisation).

Observation. La procédure est un peu plus compliquée qu'il ne le faut. Rappelons que $U[i]$ a un sens seulement si $x_i \notin Y$. L'appel à RestoreU(U) peut alors être éliminé de GenUpSets, car les valeurs significatives de U n'étaient pas modifiées par la boucle ***pour*** précédente et il n'est pas nécessaire de restituer les valeurs de U qui ne sont pas significatives.

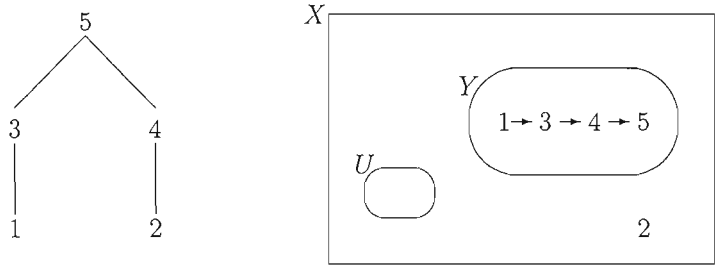

Fig. 3.2 Graphe correspondant à $\mathcal{P}$.

3.4.2 Exemple

Soit le poset de la Fig. 3.2 et trouvons $\mathcal{U}(\mathcal{P}, U, Y)$ dont la trace se trouve à la Fig. 3.3 .

Au début, nous avons :

U

1	2	3	4	5
	0			

P	1	2	3	4	5
1	1	0	1	0	1
2	0	1	0	1	1
3	0	0	1	0	1
4	0	0	0	1	1
5	0	0	0	0	1

$$P[i,j] = \begin{cases} 1 & \text{si } x_i \leq x_j \\ 0 & \text{sinon} \end{cases} \qquad \text{Pour } x_i \in X \backslash Y \qquad U[i] = \begin{cases} 1 & \text{si } x_i \in U \\ 0 & \text{si } x_i \in X \setminus (U \cup Y) \end{cases}$$

Aucune instruction pour x_i lorsque $x_i \in Y$.

3.5 Down-sets

Pour obtenir les down-sets, nous effectuons les modifications suivantes :

la ligne 6 devient : (* Générer $\mathcal{U}(\mathcal{P}, U, Y \setminus U[x])$ *) ;

la ligne 9 devient : si $(P[m,i] = 1)$ alors $U[i] \leftarrow 0$; (échange entre i et m) ;

la ligne 13 devient : (* Générer $\mathcal{U}(\mathcal{P}, U \cup D[x], Y \setminus D[x])$ *) ;

la ligne 16 devient : si $(P[j,m] = 1)$ alors $U[j] \leftarrow 1$; (échange entre j et m).

3.5.1 Exemple

Ces modifications appliquées à l'exemple 3.4.2 donnent les down-sets dont la trace se trouve à la Fig. 3.4.

3.6 Analyse de la complexité

Squire (Squire) a eu recours à l'arbre de calcul de GenUpSets, qui est un arbre pondéré dont les noeuds sont les appels à GenUpSets et où chaque appel est le fils de l'appel qui l'a généré. Le poids de chaque noeud dans cet arbre est égal au temps mis dans l'appel. Le poids d'une feuille (qui n'est pas considéré comme un noeud) est 0. Le temps total mis par GenUpSets est alors le poids total de l'arbre. Au lieu de majorer ce poids directement, le coût de chaque noeud dans l'arbre est chargé à un sous-ensemble d'autres noeuds dans l'arbre. Puis il est montré que le coût chargé à chaque noeud est $O(\log n)$.

La difficulté de l'analyse directe de GenUpSets provient du fait que l'arbre peut prendre plusieurs formes. Regardons deux formes extrêmes :

1. Supposons que $\mathcal{P}$ est une antichaîne (voir la définition 2.2) de n éléments. Dans ce cas, les listes Y' pour les appels récursifs de GenUpSets(U, Y) satisfont $|Y'| = |Y| - 1$.

Avant l'exécution, nous avons : U | 1 2 3 4 5 / 0 | $Y : 1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 5$

La trace de l'exécution :

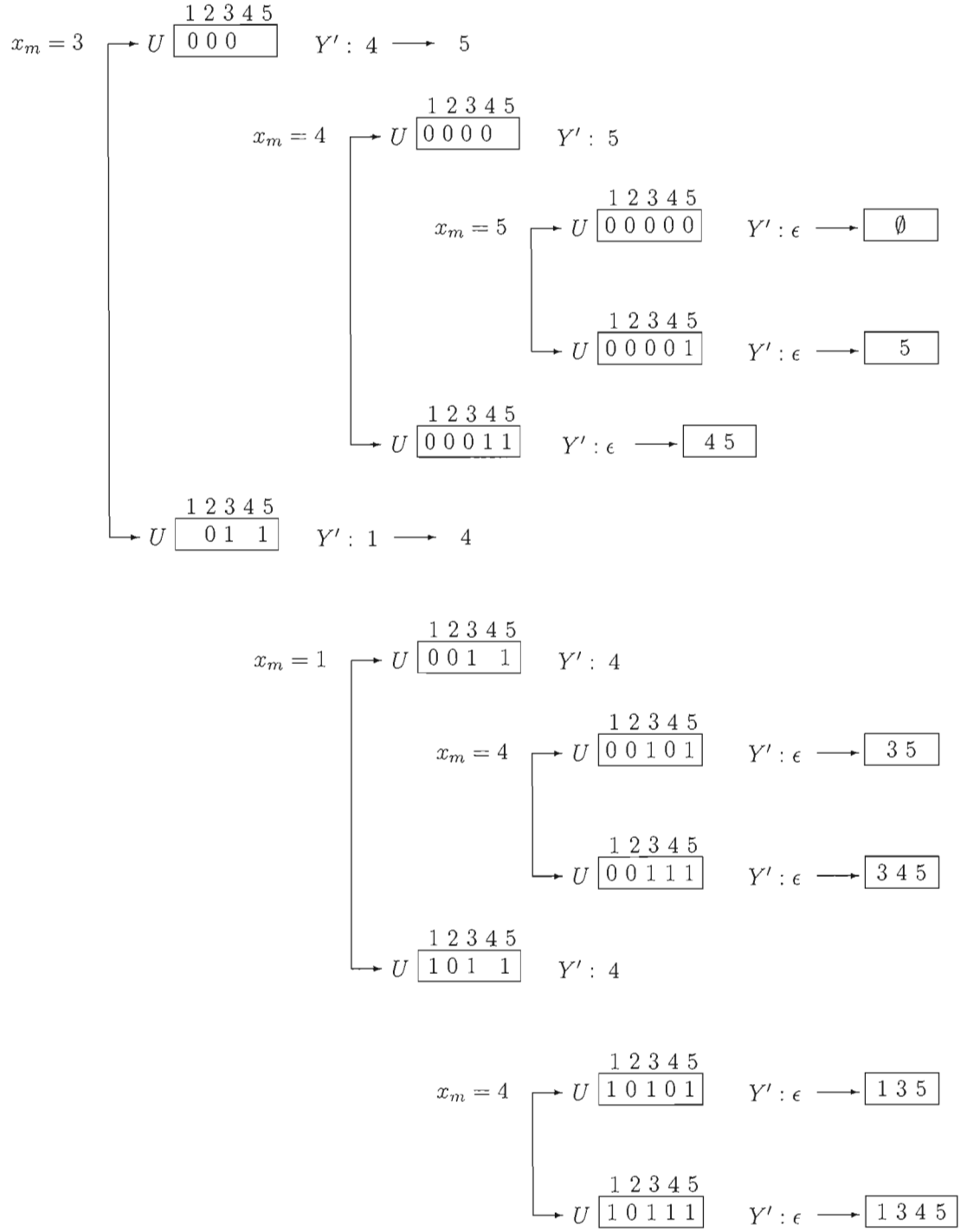

Fig. 3.3 Trace de l'algorithme de la Fig. 3.1 sur l'exemple 3.4.2.

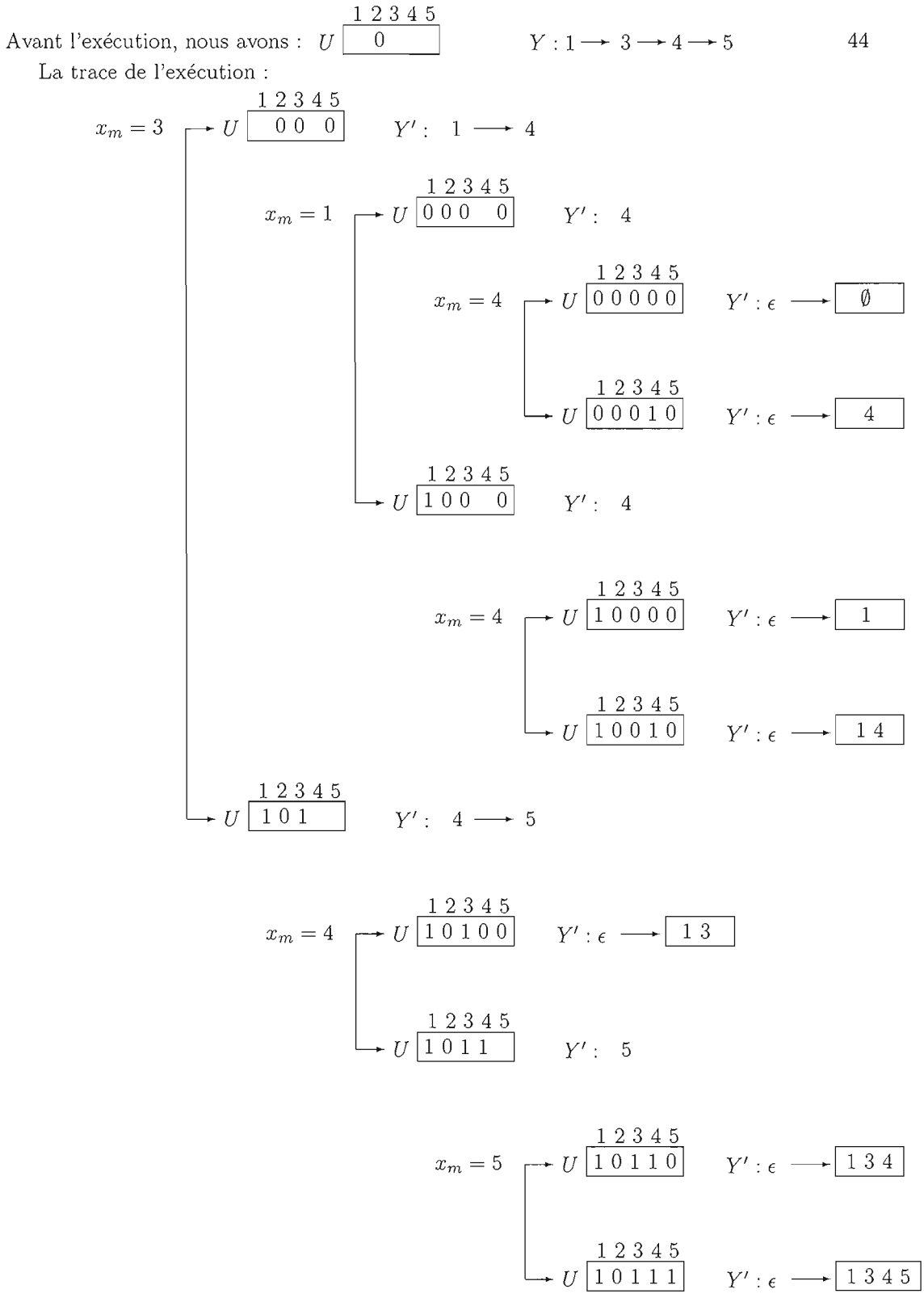

Fig. 3.4 Trace de l'algorithme de la Fig. 3.1 sur l'exemple 3.5.1.

Nous pouvons alors montrer que le poids moyen d'un noeud de l'arbre est $O(1)$ (Squire).

2. L'autre cas extrême est lorsque $\mathcal{P}$ est une chaîne de n éléments. Ici, les listes Y' satisfont $|Y'| \approx \frac{|Y|-1}{2}$. Nous pouvons alors montrer que le poids moyen d'un noeud de l'arbre sera $O(\log n)$. Les chaînes sont les pires cas de l'algorithme.

En général, la longueur des listes Y' n'est pas connue. Mais comme x_m est choisie comme élément moyen de Y, et que Y est ordonnée de façon cohérente avec R, la longueur de Y' satisfait

$$|Y| - 1 \geq |Y'| \geq \lfloor \frac{|Y| - 1}{2} \rfloor.$$

Ces bornes sur $|Y'|$ empêchent l'arbre d'être très déséquilibré et garde le poids moyen d'un noeud borné par $O(\log n)$.

Étant donné tout cela, Squire démontre que chaque noeud prend un coût de $O(\log n)$ et que le poids total de l'arbre de calcul T est $O(\log n \cdot N_T)$, où N_T est le nombre de noeuds dans T. En plus, il a démontré que T est un arbre binaire plein ; par conséquent, le nombre de noeuds dans T est deux fois moins que le nombre de feuilles. Comme l'arbre T possède une feuille par up-set de $\mathcal{P}$, le temps total pour calculer $\mathcal{U}(\mathcal{P})$ est $O(\log n \cdot u(\mathcal{P}))$.

Remarque 3.3 *Squire (Squire) a supposé dans cet algorithme que le poset $\mathcal{P}$ est donné sous forme d'une matrice P de deux dimensions et que nous disposions d'une extension linéaire du poset pour la liste originale Y. Si l'extension linéaire n'était pas disponible, nous devrions la calculer et cela prendrait un temps de $O(n^2)$. Aussi, si le poset était donné sous une autre forme, nous devrions le transformer en la matrice P de telle sorte que nous puissions mettre un temps $O(1)$ pour déterminer si $x_i \leq x_j$. Le coût d'une telle transformation dépend de la représentation originale du poset.*

CHAPITRE IV

CODE GRAY POUR LES EXTENSIONS LINÉAIRES (TRI TOPOLOGIQUE)

4.1 Motivation et littérature

L'un des plus importants ensembles associés à un poset $\mathcal{P}$ est son ensemble d'extensions linéaires $E(\mathcal{P})$. Les extensions linéaires sont très intéressantes en informatique à cause de leurs relations avec les problèmes de tri et d'ordonnancement des tâches. De même, elles sont intéressantes en combinatoire à cause de leurs relations avec les problèmes de dénombrement (Aigner, 79; Stanley, 86).

Étant donné un poset $\mathcal{P}$, deux questions se posent naturellement :

- la question de génération qui demande si les extensions linéaires $E(\mathcal{P})$ de $\mathcal{P}$ peuvent être générées efficacement ;
- la question de dénombrement qui demande si $e(\mathcal{P})$, la taille de l'ensemble $E(\mathcal{P})$, peut être determinée efficacement.

Brightwell et Winkler (Brightwell et Winkler, 92) ont prouvé que la question de dénombrement est $\#P$-complet, ce qui implique que la question de dénombrement peut être plus difficile que celle de génération. (Un problème A est dit *$\#P$-complet* si et seulement s'il est $\#P$, et tout problème dans $\#P$ peut être réduit à A en temps polynomial. $\#P$ est l'ensemble de problèmes de dénombrement associés aux problèmes de décision dans l'ensemble NP). Pruesse et Ruskey (Pruesse et Ruskey, 94) étaient les premiers à trouver un algorithme qui fonctionne en temps constant amorti (à part une petite quantité de prétraitement) pour générer des objets d'une classe combinatoire pour la-

quelle le problème de dénombrement correspondant est $\#P$-complet. Plus précisément, leur algorithme permet de générer les extensions linéaires où chaque extension diffère de sa précédente par une ou deux transpositions. L'algorithme est pratique et peut être modifié pour compter les extensions linéaires efficacement.

Leur algorithme peut être utilisé pour alterner les permutations et générer efficacement le tableau standard de Young d'une forme donnée, ainsi que n'importe quels autres objets combinatoires qui peuvent être vus comme des extensions linéaires de posets particuliers.

Le problème de génération des extensions linéaires d'un poset a été considéré par Knuth et Szwarcfiter (Knuth et Szwarcfiter, 74), Varol et Rotem (Varol et Rotem, 81) et Kalvin et Varol (Kalvin et Varol, 83). Dans ces articles, le terme « tri topologique » est utilisé au lieu du terme « extension linéaire ». L'algorithme le plus efficace parmi ces algorithmes est celui de Varol et Rotem (Varol et Rotem, 81), dont la compléxité temporelle est $O(n \cdot e(\mathcal{P}))$ (Kalvin et Varol, 83) où n est le nombre d'éléments du poset $\mathcal{P}$ et $e(\mathcal{P})$ est le nombre d'extensions linéaires. Notons que l'algorithme de Varol et Rotem est très simple, élégant et assez efficace en pratique. Le seul algorithme qui était connu pour le dénombrement des extensions linéaires d'un poset arbitraire, avant celui de Pruesse et Ruskey (Pruesse et Ruskey, 94), était celui de Wells (Wells, 71), mais il semblait difficile à analyser. Pour des classes particulières de posets, comme série-parallèle, ou de largeur bornée, des algorithmes de dénombrement étaient connus (Bouchitte et Habib, 89).

Nous allons regarder les extensions linéaires comme des permutations des éléments du poset. Quand nous générons différentes classes de permutations, le critère le plus commun (le voisinage) est que les permutations successives diffèrent par une transposition de deux de leurs éléments ; parfois c'est plus restrictif comme les transpositions d'éléments adjacents seulement. L'algorithme bien connu de Steinhaus (Steinhaus, 63), Johnson (Johnson, 63) et Trotter (Trotter, 62) fournit une énumération en ordre code Gray de toutes les $n!$ permutations de n éléments, où chaque permutation diffère de la précédente par une transposition de deux éléments adjacents. C'est pourquoi nous disons que les $n!$ permutations peuvent être générées par transpositions adjacentes. Les permutations

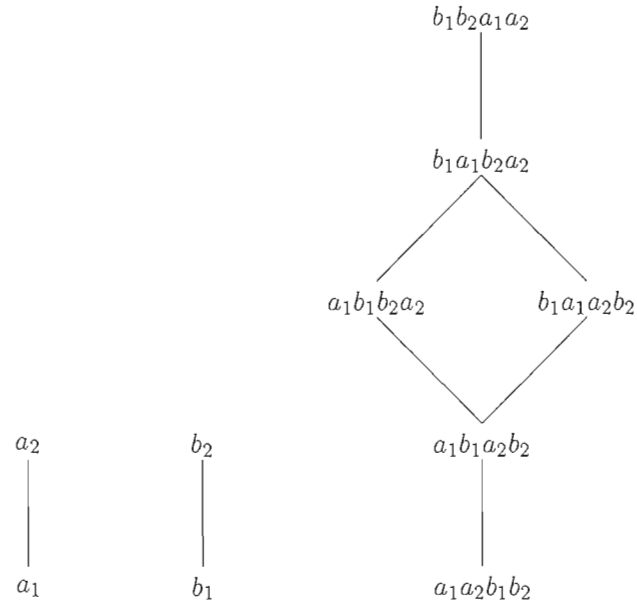

Fig. 4.1 Un poset $\mathcal{P}$ (les deux lignes verticales) et son graphe de transposition $G(\mathcal{P})$.

d'un ensemble de n éléments correspondent aux extensions linéaires du poset qui est une antichaîne (voir la définition 2.2) de n éléments.

En général, il n'est pas toujours possible de générer les extensions linéaires d'un poset par transpositions, adjacentes ou non. Par exemple, les extensions linéaires du poset constitué de deux chaînes (voir la définition 2.2) non triviales, peuvent être générées par transpositions si et seulement si n et m sont tous deux impaires (Buck et Wiedemann, 84; Eades, Hickey, et Read, 84; Ruskey, 88). Les extensions linéaires du poset de la Fig. 4.1 (deux chaînes de deux éléments chacune) ne peuvent donc pas être générées par transpositions.

Nous allons présenter l'algorithme de Pruesse et Ruskey (Pruesse et Ruskey, 94) qui génère deux fois chaque extension linéaire, et chacune est signée. L'algorithme suit la trace des signes des extensions et retourne les extensions signées positivement. Dans ce sens, l'algorithme appartient à la classe des algorithmes de génération qui génèrent plus d'objets que ceux effectivement nécessaires.

4.2 Définitions et notations

Définition 4.1 *Considérons le graphe qui a comme sommets l'ensemble $E(\mathcal{P})$ des extensions linéaires de $\mathcal{P}$, tel que deux sommets sont adjacents dans le graphe chaque fois que les extensions linéaires correspondantes diffèrent par une seule transposition. Ce graphe s'appelle le* ***graphe de transpositions*** *du poset $\mathcal{P}$ et il est noté $G(\mathcal{P})$. Le sous-graphe de $G(\mathcal{P})$ sur le même ensemble de sommets, mais contenant seulement les arêtes qui correspondent à des transpositions adjacentes, est appelé le* ***graphe de transpositions adjacentes*** *et est noté $G'(\mathcal{P})$.*

Notation 4.1 *Notons $\pm E(\mathcal{P})$, l'ensemble $\{+\ell, -\ell \mid \ell \in \mathcal{P}\}$.*

La génération des extensions linéaires de $\mathcal{P}$ par des transpositions (adjacentes) est équivalente à trouver un chemin hamiltonien dans le graphe $G(\mathcal{P})$ ($G'(\mathcal{P})$).

Exemple 4.1 *Nous voyons dans la Fig. 4.1 un poset et son graphe de transpositions.*

Les graphes de transpositions sont bipartis et connexes. Si les ensembles bipartis de $G(\mathcal{P})$ ne sont pas de la même taille, alors le graphe ne possède pas de cycle hamiltonien. Si la différence entre les tailles de ces ensembles est > 1, alors le graphe ne possède pas de chemin hamiltonien et par conséquent, les extensions linéaires de $\mathcal{P}$ ne peuvent pas être générées par transpositions (Pruesse et Ruskey, 94).

Exemple 4.2 *Dans la Fig. 4.1, nous voyons que les tailles des ensembles de bipartition diffèrent par deux. Ainsi, les extensions linéaires du poset ne peuvent pas être générées par transpositions.*

Définition 4.2 *Si toute paire d'éléments de $S(\mathcal{P})$ est comparable (voir la définition 2.1), alors $\mathcal{P}$ est un* ***ordre total****.*

Notation 4.2 *Si $\mathcal{P}$ est un ordre total sur $S(\mathcal{P}) = \{x_1, x_2, \ldots, x_n\}$ tel que $x_i \prec x_j$ si et seulement si $i < j$, alors nous utilisons $x_1 x_2 \ldots x_n$ pour désigner $\mathcal{P}$.*

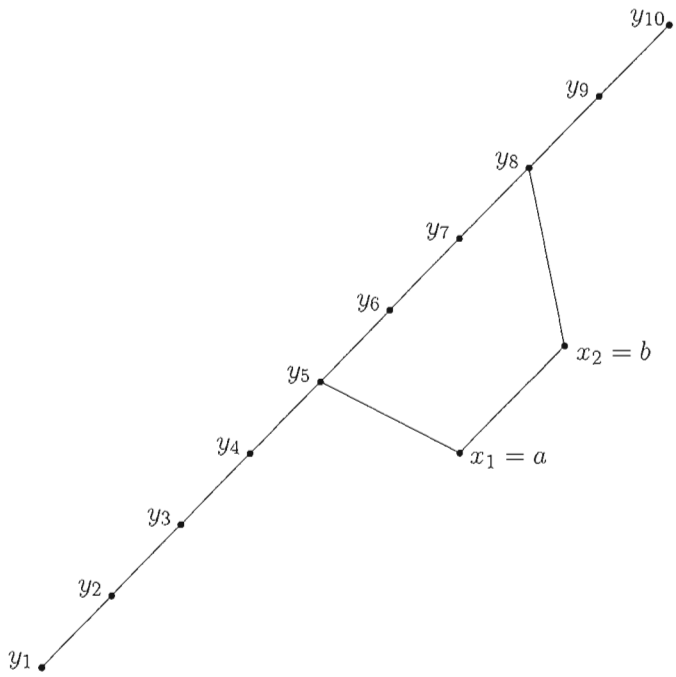

Fig. 4.2 Un B-poset.

Définition 4.3 *Un* ***B-poset*** *est un poset $\mathcal{P}$ dont les éléments peuvent être partitionnés en deux chaînes disjointes $x_1 \prec x_2 \prec \cdots \prec x_n$ et $y_1 \prec y_2 \prec \cdots \prec y_m$ telles que $y_j \not\preceq x_i$ pour tout i et j, où $1 \leq i \leq n$ et $1 \leq j \leq m$. Lorsque $n = 2$, nous l'appelons* ***2B-poset****.*

Exemple 4.3 *Voir le B-poset de la Fig. 4.2.*

Notation 4.3 *Notons $mr(x_i)$, le plus grand indice j tel que $x_i \parallel x_j$; si $x_i < y_1$, alors $mr(x_i) = 0$.*

Pour la Fig. 4.2, nous avons $mr(a) = 4$ et $mr(b) = 7$. Nous avons déjà vu la définition de $G \times K_2$ (voir la Fig. 1.4), i.e., la multiplication d'un graphe par une arête. Le lemme suivant va nous être utile pour la suite.

4.3 $G'(\mathcal{P}) \times K_2$ est hamiltonien

Lemme 4.1 *Si a et b sont deux frères (voir la définition 2.7) dans $\mathcal{P}$, alors $G(\mathcal{P}) \cong G(\mathcal{P}+ab) \times K_2$.*

Preuve : Remarquons que $E(\mathcal{P}) = E(\mathcal{P}+ab) \cup E(\mathcal{P}+ba)$. Toute extension linéaire ℓ de $\mathcal{P}$ qui transpose a et b dans ℓ produit une autre extension linéaire de $\mathcal{P}$. Par la suite, l'opération qui transpose a et b dans une extension linéaire fournit un isomorphisme entre $G(\mathcal{P}+ab)$ et $G(\mathcal{P}+ba)$. □

Si $e(\mathcal{P}) = 1$ (i.e., si $\mathcal{P}$ est un ordre total), alors $G \times K_2$ est une arête. Dans le but de démontrer par récurrence l'existence de cycles hamiltoniens, nous considérons que ce graphe possède un cycle hamiltonien ; puisqu'il a un chemin hamiltonien dont les deux extrémités sont adjacentes, alors il a un cycle hamiltonien.

L'algorithme de Pruesse et Ruskey (Pruesse et Ruskey, 94) que nous allons présenter dans ce chapitre est basé sur la preuve que $G'(\mathcal{P}) \times K_2$ est hamiltonien. Pruesse et Ruskey (Pruesse et Ruskey, 91) ont montré cela dans le cas particulier du poset appelé B-poset (voir la définition 4.3).

Remarque 4.1 *Notons que $k = x_1x_2\cdots x_ny_1y_2\cdots y_m$ est une extension linéaire de n'importe quel B-poset. Nous l'appelons l'**extension linéaire canonique** d'un B-poset.*

Lemme 4.2 *(*prouvé dans (Pruesse et Ruskey, 91)*) Soit $\mathcal{P}$ un B-poset. Alors il existe un cycle hamiltonien dans $G'(\mathcal{P}) \times K_2$ passant par l'arête $[+k, -k]$.*

Le graphe $G'(\mathcal{P}) \times K_2$, où $\mathcal{P}$ est le B-poset de la Fig. 4.2, est celui de la Fig. 4.3.
Les arêtes correspondantes à l'isomorphisme entre deux copies de $G'(\mathcal{P})$ ont été omises pour plus de clarté. Nous pouvons penser que nous passons à travers une arête verticale vers le haut lorsque $b = x_2$ est transposé avec son voisin de droite, et à travers une arête horizontale lorsque $a = x_1$ est transposé avec l'un de ses voisins.

Le cycle hamiltonien entre $+k$ et $-k$ est dans la Fig. 4.4. Le cycle passe par l'arc lorsque $mr(b)$ est pair.

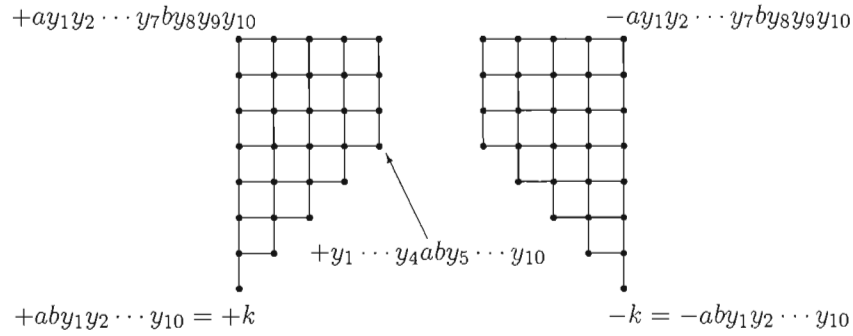

Fig. 4.3 Graphe $G'(\mathcal{P})\times K_2$ du B-poset $\mathcal{P}$ de la Fig. 4.2.

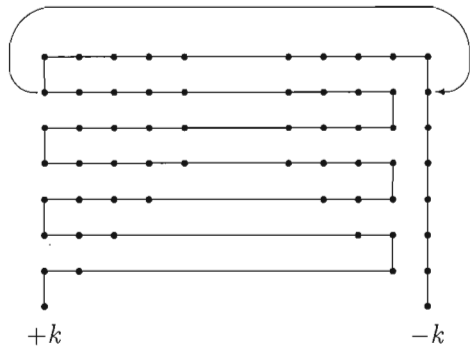

Fig. 4.4 Un cycle hamiltonien dans le graphe $G'(\mathcal{P})\times K_2$ de la Fig. 4.3.

Définition 4.4 *Nous obtenons un graphe similaire à celui de la Fig. 4.3 chaque fois que $a \parallel y_1$; nous disons que nous avons un* ***cas typique****. Si $a \prec y_1$, alors $G'(\mathcal{P})$ est un chemin ; nous disons que nous avons un* ***cas atypique****. Autrement dit, le cas typique survient lorsque $mr(a) > 0$ et le cas atypique survient lorsque $mr(a) = 0$.*

Théorème 4.1 *Pour chaque poset $\mathcal{P}$, le graphe $G'(\mathcal{P}) \times K_2$ est hamiltonien.*

Preuve : La preuve se fait par récurrence sur $|S(\mathcal{P})|$ (Pruesse et Ruskey, 94). □

Remarque 4.2 *Les B-posets utilisés dans ce chapitre sont tous des $2B$-posets. Dans le cas typique, le cycle de la Fig. 4.4 peut être utilisé ; dans le cas atypique, le cycle est clair (nous déplaçons b à droite le plus loin possible, nous changeons de signe, puis nous déplaçons b à gauche). Si $mr(b)$ est pair, le cycle de la Fig. 4.4 est un peu différent et utilise l'arête $[+a\gamma b, -a\gamma b]$, où $\gamma = y_1 y_2 \cdots y_m$ (voir la flèche). Ces cycles sont utilisés dans l'algorithme.*

Corollaire 4.1 *Si $\mathcal{P}$ est un poset avec une paire de frères, alors $G(\mathcal{P})$ est hamiltonien.*

Preuve : Supposons que $\mathcal{P}$ possède une paire de frères a et b. Selon le théorème 4.1, $G'(\mathcal{P}) \times K_2$ est hamiltonien ; par conséquent, $G(\mathcal{P}) \times K_2$ est hamiltonien. D'où, selon le lemme 4.1, $G(\mathcal{P})$ est hamiltonien. □

Notation 4.4 *Pour $T \subseteq S(\mathcal{P})$, nous notons $\mathcal{P} \setminus T$, le poset sur l'ensemble $S(\mathcal{P}) \setminus T$ ayant comme relation l'ensemble $R(\mathcal{P}) \cap (S(\mathcal{P}) \setminus T)^2$.*

Notation 4.5 *Pour les posets $\mathcal{P}$ et $\mathcal{Q}$, si $R(\mathcal{P}) \cup R(\mathcal{Q})$ est antisymétrique, nous notons $\mathcal{P} + \mathcal{Q}$ le poset sur l'ensemble $S(\mathcal{P}) \cup S(\mathcal{Q})$ ayant comme relation la clôture transitive de $R(\mathcal{P}) \cup R(\mathcal{Q})$.*

Exemple 4.4 *Par exemple, $\mathcal{P} + abc$ est le poset sur l'ensemble $S(\mathcal{P}) \cup \{a, b, c\}$ ayant comme relation la clôture transitive de $R(\mathcal{P}) \cup \{(a, b), (b, c)\}$. Si $\mathcal{P} + \mathcal{Q} = \mathcal{P}$, alors nous disons que $\mathcal{P}$ induit $\mathcal{Q}$. Par exemple, si $\mathcal{P} + abc = \mathcal{P}$, alors $\{(a, b), (b, c)\} \subseteq R(\mathcal{P})$ et toute extension linéaire de $\mathcal{P}$ vérifie $a \prec b \prec c$; par conséquent, nous disons que $\mathcal{P}$ **induit** abc.*

4.4 Algorithme

4.4.1 Idée de l'algorithme

Le théorème 4.1 est constructif et il est la base de l'algorithme qui fonctionne en temps constant amorti, i.e., il génère toutes les extensions linéaires d'un poset $\mathcal{P}$ en $O(e(\mathcal{P}))$. C'est un algorithme sur place ; il maintient un tableau *le* qui contient l'extension linéaire courante, et une variable $IsPlus$ qui suit la trace du signe (+ ou −). Nous passons d'une extension linéaire à la suivante en faisant des changements au tableau *le* ou en reversant le signe.

La procédure principale que nous appelons GenLE (Fig. 4.6) et qui est tirée de (Pruesse et Ruskey, 94) est récursive et suit le chemin indiqué dans la Fig. 4.4. Chaque niveau de la récursivité possède une paire d'éléments minimaux du sous-poset courant. Par exemple, dans le poset de la Fig. 4.1, (a_1, b_1) est la paire d'éléments minimaux de $\mathcal{P}_1 = \mathcal{P}$ et (a_2, b_2) est la paire d'éléments minimaux de $\mathcal{P}_2 = \mathcal{P}_1 \setminus \{a_1, b_1\}$. Ces paires sont déterminées par un prétraitement qui est le suivant.
Nous supprimons récursivement des paires (a_i, b_i) d'éléments minimaux pour $i = 1, 2, \cdots$ jusqu'à épuisement. Si un seul élément minimal est rencontré, nous l'éliminons tout simplement et il ne fait pas partie d'une paire. Soit MaxPair, la dernière paire d'éléments minimaux supprimée de $\mathcal{P}$; le reste de $\mathcal{P}$ forme un ordre total ou l'ensemble vide. Ce prétraitement est détaillé à la Fig. 4.5.
Notons que MaxPair n'est pas déterminé d'une façon unique par le poset, car il est lié à l'ordre d'éléments enlevés de $\mathcal{P}$.

Définition 4.5 *Nous disons qu'une extension linéaire ℓ est* ***en ordre propre jusqu'à*** *i si pour tout $1 \leq j \leq i$, les éléments a_j et b_j sont adjacents dans ℓ et si ℓ induit les ordres $a_1 a_2 \cdots a_i a_h$ et $a_1 a_2 \cdots a_i b_h$ pour tout h, où $i < h \leq$* MaxPair.

Remarque 4.3 *L'extension linéaire initiale de la liste doit être en ordre propre jusqu'à* MaxPair *; le prétraitement de la Fig. 4.5 l'assure.*

$i \leftarrow j \leftarrow 0$;

$\mathcal{Q} \leftarrow \mathcal{P}$;

tant que $S(\mathcal{Q}) \neq \emptyset$ **faire**

  si $\mathcal{Q}$ possède exactement un élément minimal x **alors**

    $j \leftarrow j + 1$;

    $le[j] \leftarrow x$;

    $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{x\}$;

  sinon

    soient a', b' deux éléments minimaux quelconques de $\mathcal{Q}'$

    $i \leftarrow i + 1$;

    $j \leftarrow j + 2$;

    $a[i] \leftarrow a'$;

    $b[i] \leftarrow b'$;

    $le[j-1] \leftarrow a'$;

    $le[j] \leftarrow b'$;

    $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{a', b'\}$;

  fin si ;

fin tant que ;

MaxPair $\leftarrow i$;

Fig. 4.5 Prétraitement

```
procedure GenLE ( i : integer );
var mrb, mra, mla, x : integer; typical : boolean;
begin
if i > 0 then begin
  GenLE (i − 1);
  mrb := 0; typical := false;
  while Right(b[i]) do begin
    mrb := mrb + 1;
    Move(b[i], right); GenLE(i − 1);
    mra := 0;
    if Right(a[i]) then begin
      typical := true;
      repeat
        mra := mra + 1;
        Move(a[i], right); GenLE(i − 1);
      until not Right(a[i]);
    end {if};
    if typical then begin
      Switch(i − 1); GenLE(i − 1);
      if odd(mrb) then mla := mra − 1 else mla := mra + 1;
      for x := 1 to mla do begin
        Move(a[i], left); GenLE(i − 1);
      end;
    end {if};
  end {while};
  if typical and odd (mrb)
    then Move(a[i], left);
    else Switch(i − 1);
  GenLE(i − 1);
  for x := 1 to mrb do begin
    Move(b[i], left); GenLE(i − 1);
  end;
end {if};
end {GenLE};
```

Fig. 4.6 Procédure GenLE

4.4.2 Implantation de l'algorithme

L'implantation de l'algorithme maintient quatre tableaux globaux : le tableau le qui est l'extension linéaire, son inverse li, les tableaux a et b qui contiennent les éléments a_i et b_i de telle sorte que $a[i]$ contient toujours la valeur gauche de la i-ième paire et $b[i]$ celle de droite. Aussi, le signe courant est maintenu.

La fonction booléenne Right(x) est utilisée pour déterminer si l'élément x peut se déplacer vers la droite. Elle fonctionne en $O(1)$ comme suit :

- Right($b[i]$) : Retourne vrai seulement si $b[i]$ est incomparable avec l'élément à sa droite dans le tableau le.
- Right($a[i]$) : Retourne vrai seulement si $a[i]$ est incomparable avec l'élément à sa droite dans le tableau le et l'élément à sa droite n'est pas $b[i]$.

Dans un vrai code, ces deux fonctions utilisent plus de paramètres pour pouvoir chercher x dans le et décider s'il s'agit de Right($b[i]$) ou de Right($a[i]$).

Les procédures Move et Switch sont utilisées pour changer l'extension linéaire courante. Elles fonctionnent en $O(1)$ comme suit :

- Switch(i) : Si $i = 0$, alors le signe est changé, i.e., IsPlus est changée. Si $i > 0$, alors a_i et b_i sont transposés.
- Move(x,left) : Cet appel transpose x avec l'élément à sa gauche.
- Move(x,right) : Cet appel transpose x avec l'élément à sa droite.

Chaque fois qu'une nouvelle extension linéaire ℓ de $\mathcal{P}_i$ est générée par l'appel GenLE(i) (i.e., chaque fois que Move ou Switch est appelée), GenLE($i-1$) est appelée. L'appel GenLE($i-1$) déplace $a_1, a_2, \cdots, a_{i-1}, b_{i-1}$ dans toutes les positions possibles à travers ℓ en gardant l'ordre $a_{i-1} \prec b_{i-1}$ (ou $b_{i-1} \prec a_{i-1}$, selon leur ordre au moment de l'appel de GenLE($i-1$)). Si $i = 1$, alors GenLE($i-1$) ne fait rien.

En supposant que Right(b[MaxPair+1]) est fausse, l'appel initial à la procédure principale est GenLE(MaxPair+1). Cet appel consiste exactement des appels suivants, que nous appelons la **suite d'appels** :

GenLE(MaxPair) ; Switch(MaxPair) ; GenLE(MaxPair) ;

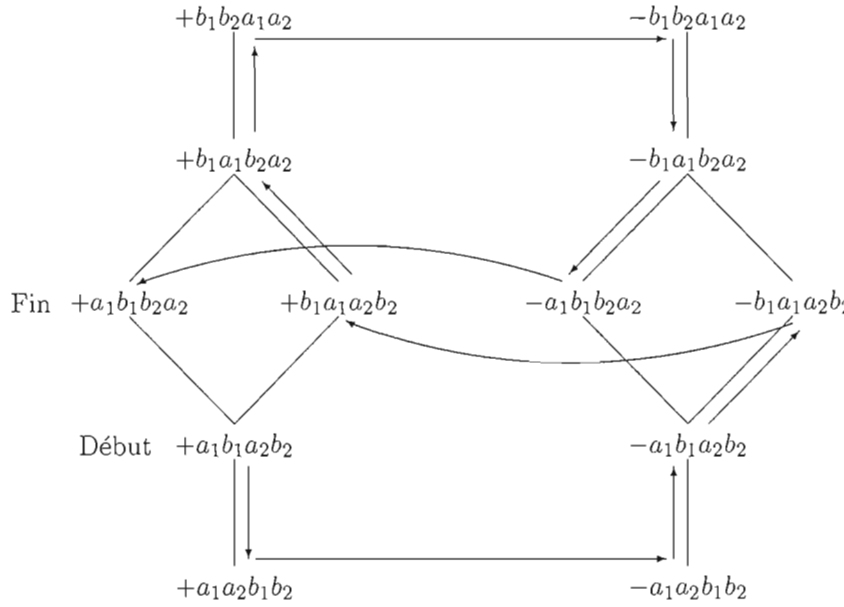

Fig. 4.7 Un cycle hamiltonien dans le graphe $G'(\mathcal{P}) \times K_2$ où $G(\mathcal{P})$ est le graphe de la Fig. 4.1.

Fonctionnement. L'algorithme consiste à exécuter le prétraitement, à initialiser IsPlus à +, puis à exécuter la suite d'appels.

4.4.3 Exemple

Si nous prenons l'exemple de la Fig. 4.1, le prétraitement nous retourne $a_1b_1a_2b_2$, et en mettant IsPlus à +, nous commençons avec $+a_1b_1a_2b_2$ et nous exécutons la suite d'appels : GenLE(2) ; Switch(2) ; GenLE(2).
Après l'exécution de la procédure principale de la Fig. 4.6, nous obtenons le résultat dans le Tab. 4.1 correspondant au graphe de la Fig. 4.7.

Théorème 4.2 *L'algorithme GenLE de la Fig. 4.6 génère les extensions linéaires selon un chemin hamiltonien dans* $G'(\mathcal{P}) \times K_2$.

Preuve : voir (Pruesse et Ruskey, 94).

Procédure	**Extension linéaire**	
	$+a_1b_1a_2b_2$	(* initialisation *)
GenLE(2)		
GenLE(1)		
Move(b_1, right)	$+a_1a_2b_1b_2$	
Switch(0)	$-a_1a_2b_1b_2$	
Move(b_1, left)	$-a_1b_1a_2b_2$	
Switch(1)	$-b_1a_1a_2b_2$	
GenLE(1)		
Switch(0)	$+b_1a_1a_2b_2$	
Switch(2)	$+b_1a_1b_2a_2$	
GenLE(2)		
GenLE(1)		
Move(a_1, right)	$+b_1b_2a_1a_2$	
Switch(0)	$-b_1b_2a_1a_2$	
Move(a_1, left)	$-b_1a_1b_2a_2$	
Switch(1)	$-a_1b_1b_2a_2$	
GenLE(1)		
Switch(0)	$+a_1b_1b_2a_2$	

Tab. 4.1 Trace de la procédure GenLE sur l'exemple du poset de la Fig. 4.1 qui correspond au graphe de la Fig. 4.7.

4.5 Analyse de la complexité

Dans cette analyse, nous supposons que Right, Switch et Move peuvent être implantées en temps constant. Ces implantations sont facilement accomplies tant que l'inverse *li* de *le* est maintenu. Chaque appel à Move et à Switch génère une extension linéaire; de plus, remarquons que l'appel GenLE(i) génère au moins deux appels de GenLE(i – 1). Chaque itération de la boucle *while* ou *for* de l'algorithme exécute un Move et par conséquent, génère une extension linéaire. Le seul appel dans lequel GenLE est récursivement appelé, sans qu'une extension linéaire ne soit générée, est lorsque $i = 0$, et ceci arrive au plus une fois par extension linéaire générée. Par conséquent, l'algorithme fonctionne en temps constant par extension linéaire en générant $\pm E(\mathcal{P})$. En supprimant les extensions linéaires précédées par –, nous générons $E(\mathcal{P})$ en temps constant amorti.

4.6 Code Gray

Nous allons montrer que les extensions linéaires peuvent être générées de telle sorte que les extensions successives diffèrent par une ou deux transpositions adjacentes.

Notation 4.6 *Si α et β sont des extensions de $\mathcal{P}$, alors nous notons $D(\alpha, \beta)$, la **distance** de α à β dans $G(\mathcal{P})$ et $D'(\alpha, \beta)$, celle dans $G'(\mathcal{P})$.*

Définition 4.6 *Nous disons qu'un ordre $\alpha_1, \cdots, \alpha_{e(\mathcal{P})}$ des extensions linéaires de $\mathcal{P}$ possède un **délai** c, si $D'(\alpha_i, \alpha_{i+1}) \leq c$ pour tout $0 \leq i < e(\mathcal{P})$, où $\alpha_0 = \alpha_{e(\mathcal{P})}$.*

Nous allons montrer l'existence d'un ordre de $E(\mathcal{P})$ de délai 2 trouvé en temps constant amorti.

Lemme 4.3 *Si G est biparti et $G \times K_2$ est hamiltonien, alors G^2 est hamiltonien.*

Preuve : voir (Pruesse et Ruskey, 94).

Théorème 4.3 *Les extensions linéaires de n'importe quel poset peuvent être générées avec un délai 2 en temps constant amorti.*

Preuve : Nous exécutons l'algorithme GenLE de la Fig. 4.6, mais au lieu de supprimer les extensions linéaires précédées par –, nous supprimons une fois toute les deux extensions linéaires; i.e., si nous générons la liste $l_1, l_2, \cdots, l_5, \cdots$, alors nous retournons la liste $l_1, l_3, l_5, \cdots$. Selon le lemme 4.3, cet ordre possède un délai de 2. Il a par conséquent la même complexité que GenLE, i.e., un temps contant amorti. □

Conclusion. Pruesse et Ruskey (Pruesse et Ruskey, 94) ont trouvé un algorithme pour générer les extensions linéaires d'un poset en temps constant amorti. C'est une amélioration de l'algorithme de (Kalvin et Varol, 83) qui s'exécute en $O(n)$. De plus, ils ont suggéré de générer ces extensions linéaires sans boucle; c'est ce que nous allons voir dans le chapitre suivant.

CHAPITRE V

IMPLANTATION SANS BOUCLE DU CODE GRAY DES EXTENSIONS LINÉAIRES D'UN POSET

5.1 Introduction

Nous avons vu dans le chapitre précédent que Pruesse et Ruskey (Pruesse et Ruskey, 94) ont trouvé un algorithme pour générer les extensions linéaires d'un poset en temps constant amorti. Canfield et Williamson (Canfield et Williamson, 95) ont étendu leur travail en trouvant le premier algorithme sans boucle pour la génération des extensions linéaires d'un poset. Leur algorithme utilise une transposition adjacente comme opération de base pour générer l'extension linéaire suivante. En effet, chaque extension linéaire est générée deux fois, mais seulement une des deux extensions est retenue pour le résultat final. Comme l'algorithme génère toutes les extensions signées, avec chaque extension apparaissant quelque part avec chaque signe, alors chaque extension diffère de la précédente par deux transpositions adjacentes au plus.

Korsh et Lafollette (Korsh et Lafollette, 2002) ont trouvé un autre algorithme pour générer les extensions linéaires d'un poset. Leur algorithme utilise un déplacement comme opération de base pour générer l'extension linéaire suivante. Par définition, un ***déplacement*** requiert qu'un entier soit déplacé vers la gauche dans l'extension. Lorsque le déplacement est d'une position vers la gauche, le résultat est le même qu'une transposition adjacente, alors que lorsque le déplacement est de plus d'une position vers la gauche, il est équivalent à plus d'une transposition adjacente. Leur algorithme, que nous allons présenter, génère chaque extension linéaire une seule fois.

5.2 Version récursive de l'algorithme

Korsh et Lafollette (Korsh et Lafollette, 2002) ont donné l'algorithme récusif : algorithme 1 de la Fig. 5.1 commençant avec une extension linéaire pour ensuite générer toutes les autres.

Pour générer les autres extensions linéaires :

si n = 1, **alors**
  l'extension initiale est la seule extension
sinon
  générer les autres extensions commençant par le premier entier de l'extension initiale
  pour tout autre entier qui peut être placé au début d'une extension
    déplacer cet entier au début et
    générer les autres extensions commençant par cet entier
  fin pour
fin si

Fig. 5.1 Algorithme 1 pour générer le reste des extensions.

5.2.1 Exemple

Par exemple, si $n = 4$ et s'il n'y a pas de contraintes sur les entiers du poset, alors tous les entiers $4, 3, 2, 1$ peuvent être au début de l'extension. Remarquons qu'une extension se lit de gauche à droite. L'algorithme 1 de la Fig. 5.1 peut générer à partir de 4321, les 24 extensions linéaires du Tab. 5.1.

Remarque 5.1 *L'ordre actuel des extensions linéaires est déterminé par l'extension initiale et par la sélection de l'entier qui sera au début de l'extension suivante. Le déplacement d'un élément au début de l'extension se fait en enlevant cet élément et en le plaçant au début.*

1. 4321,	7. 2431,	13. 1234,	19. 3124,
2. 4312,	8. 2413,	14. 1243,	20. 3142,
3. 4132,	9. 2143,	15. 1423,	21. 3412,
4. 4123,	10. 2134,	16. 1432,	22. 3421,
5. 4213,	11. 2314,	17. 1342,	23. 3241,
6. 4231,	12. 2341,	18. 1324,	24. 3214.

Tab. 5.1 Trace de l'algorithme 1 (Fig. 5.1) quand $n = 4$ et l'extension initiale est 4321.

5.2.2 Fonctionnement de l'algorithme

L'algorithme 1 de la Fig. 5.1 fonctionne comme suit : il commence avec une extension initiale et exécute une suite de déplacements. Chaque déplacement produit l'extension suivante. En plus, chaque déplacement fait intervenir le début (position) d'une sous-extension dont un entier de droite est déplacé à cette position.
Par exemple, l'extension 2 est obtenue de l'extension 1 en déplaçant le 1 à la position 3 et l'extension 13 est obtenue de l'extension 12 en déplaçant le 1 à la position 1.

5.3 Version non-récursive de l'algorithme

Afin de trouver une version non-récursive et sans boucle (Korsh et Lafollette, 2002), Korsh et Lafollette ont constaté nécessaire de déterminer l'entier à être déplacé et la position à laquelle il doit être déplacé, et de faire ce déplacement en temps constant.

Remarque 5.2 *Quand l'algorithme 1 de la Fig. 5.1 s'exécute, un déplacement à la position k est fait une fois que toutes les sous-extensions qui font intervenir des entiers en position $k+1$ à n aient été générées. Par exemple, le déplacement à la position 2 dans l'extension 3 du Tab. 5.1 survient, car toutes les sous-extensions de 2 et 1 ont été générées. Afin de connaître quels sont les entiers qui peuvent être déplacés à la position k, **nous associons une liste de tels entiers à la position** k, et nous la notons **Lists**. Le lemme 5.1 permet de préciser ces listes.*

Lemme 5.1 *Soit $S = i_k, i_{k+1}, \cdots, i_n$, la première sous-extension générée par l'algorithme 1 de la Fig. 5.1 qui fait intervenir ces entiers. Chaque entier, différent de i_k, qui puisse être au début, position k, d'une sous-extension faisant intervenir ces entiers apparaîtra en position $k+1$ de la sous-extension une fois que toutes les sous-extensions commençant par i_k aient été générées par cet algorithme.*

Preuve : Si un entier n'apparaît pas en position $k+1$, c'est parce que les contraintes de l'ordre partiel l'ont empêché d'atteindre cette position. Ces mêmes contraintes vont aussi l'empêcher d'atteindre la position k. □

Exemple 5.1 *Dans notre exemple, Tab. 5.1, si $S = 4321$, alors $k = 1$. Les seuls entiers, différents de 4, qui peuvent être au début d'une sous-extension faisant intervenir 4, 3, 2 et 1 sont 3, 2 et 1. Comme toutes les sous-extensions faisant intervenir 3, 2 et 1 sont générées dans les premières six extensions, chacun de ces entiers apparaît en position $k+1 = 2$.*
Si $S = 234$, alors $k = 2$. Les seuls entiers, différents de 2, qui peuvent être au début d'une sous-extension faisant intervenir 2 , 3 et 4 sont 3 et 4 . Comme toutes les sous-extensions faisant intervenir 3 et 4 sont générées dans les extensions 13 et 14, chacun de ces entiers apparaît en position $k+1 = 3$.

Remarque 5.3 *En général, les entiers qui apparaissent en position $k+1$ ne peuvent pas tous être en position k, car l'ordre partiel peut empêcher certains d'atteindre la position k.*

Le lemme 5.1 est important puisqu'il nous permet de contrôler les entiers qui apparaissent en position $k+1$ quand l'algorithme 1 (Fig. 5.1) s'exécute pour déterminer ceux qui peuvent atteindre la position k.

5.4 Implantation et fonctionnement de l'algorithme

Quand un entier en position $k+j$ se déplace en position k, chacun des entiers en positions $k, k+1, \cdots, k+j-1$ se déplace par une seule position à droite. Ce déplacement à gauche et les déplacements à droite forment un déplacement circulaire, qui sera désigné dans l'algorithme par *déplacer* $k+j$ *à* k.

Une fois que toutes les extensions faisant intervenir les entiers en positions $k+1$ à n sont générées, l'entier initial (à la position k) et les autres qui ont atteint la position $k+1$ sont les seuls *candidats pour la liste associée à la position* k. Ces listes ne sont pas toujours complètes, mais seront mises à jour au cours de l'exécution.

Quand un entier de la liste, associée à la position k, est déplacé à la position k, il est supprimé de la liste. Nous disons qu'une *position* est *finie* quand sa liste devient vide. À ce moment là, la liste de son voisin gauche devient *fermée* si elle n'est pas vide, sinon elle devient *non fermée.*

Un *drapeau* est maintenu pour indiquer si une liste est *fermée* ou *non fermée.*

Remarque 5.4 *En général, nous pouvons avoir plusieurs positions auxquelles il reste des entiers à déplacer ; pourtant, nous commençons toujours par celle la plus à droite. Dans notre exemple (Tab. 5.1), dans l'extension 13, les positions 1, 2, et 3 contiennent toutes des entiers à être déplacés à ces positions, mais la position la plus à droite, position 3, est la position suivante, à laquelle un entier est déplacé pour générer l'extension suivante 14.*

Afin de trouver la position suivante à laquelle nous déplaçons un entier, en temps constant, une liste *prête* et une liste *finie* sont maintenues. La liste *prête* contient toutes les positions auxquelles il y a des entiers à déplacer. De plus, chaque fois qu'une telle position possède des positions successives à sa gauche qui contiennent des entiers, chacun est contraint d'être avant celui à sa droite ; alors chacun de ces entiers est aussi placé dans la liste *prête.* Ces entiers successifs forment une *chaîne.*

Les premier et dernier éléments d'une telle chaîne contiendront toujours l'information sur la localisation du dernier élément de la chaîne (le plus à gauche) et du premier

élément (le plus à droite).

La liste *finie* est similaire, mais elle contient l'information sur ces positions « *finies* »qui sont finies. Elle contient aussi une chaîne par position, comme la liste *prête*.

Les deux listes *prête* et *finie* contiennent les positions en ordre, avec la position la plus à droite mise au début de la liste et celle la plus à gauche mise à la fin de la liste. Toutes les positions sont exactement dans l'une de ces deux listes en tout temps.

5.4.1 Phrases clés pour l'algorithme

Après toutes ces explications et avant de présenter l'algorithme, nous avons besoin d'expliquer certaines phrases utilisées dans l'algorithme.

Notation 5.1 *Notons $I(x)$, l'entier en position x, et xlt (respectivement xrt) la position $x-1$ (respectivement $x+1$).*

Mettre à jour la liste de l'élément pointé par p, et celle pointé par plt

1. *Ajouter $I(p)$ à la liste de plt (et mettre la liste de plt à non fermée) :*
   cela arrive quand plt est le successeur de p sur la liste *prête*, la liste de plt est vide et $I(p)$ peut être avant $I(plt)$.
2. *Ajouter $I(prt)$ à la liste de p :*
   cela arrive quand la liste de p est vide.
3. *Ajouter $I(p+j)$ à la liste de plt :*
   cela arrive quand la liste de plt est *non fermée*, $I(p+j)$ peut être avant $I(plt)$ et plt est le successeur de p dans la liste *prête*.

Mettre à jour le drapeau de l'élément pointé par plt

Quand p est *fini*, le drapeau de plt est mis à *non fermé* si sa liste est vide et à *fermé* sinon. Le drapeau de prt est toujours mis à *non fermé*.

5.4.2 Initialisation

Au début de l'exécution de l'algorithme 2 (Fig. 5.2) sur l'extension initiale, la chaîne la plus à droite est sur la liste *finie*. Toutes les autres chaînes sont sur la liste *prête*. Toutes les listes sont vides et tous les drapeaux sont *non fermés*.

5.5 Algorithme sans boucle

Les structures de données suivantes permettent de réaliser l'algorithme sans boucle.
L'extension est représentée par une liste doublement chaînée, appelée *sort* ; un *élément* de *sort* est un enregistrement qui contient un pointeur sur son voisin gauche, un pointeur sur son voisin droite et l'entier à sa position dans l'extension. *Sort* pointe sur le début de la liste et *end* sur la fin. Cela permet que le déplacement soit fait en temps constant. Les listes *prête* et *finie*, représentées respectivement par m et f, contiennent aussi des éléments de *sort*.
Nous utilisons une matrice à deux dimensions, *less*, pour stocker les contraintes sur les entiers. Pour $1 \leq i, j \leq n$, $less[i][j]$ est vrai si i est contraint d'être avant j et faux sinon.
Flag, Last, First, Pred, Next et *Lists* sont des tableaux. Par l'élément i nous voulons dire l'élément de *sort* qui contient l'entier i. $Flag[i]$ est *fermé* si l'élément i est *fermé* et *non fermé* sinon.
$Pred[i]$ et $Next[i]$ contiennent, respectivement, des pointeurs sur les éléments prédécesseurs et successeurs de i dans m et f. *Pred* et *Next* font que les listes *prête* et *finie* sont deux listes doublement chaînées.
$Last[i]$ et $First[i]$ contiennent, respectivement, des pointeurs au dernier et premier éléments d'une chaîne dans m ou f quand i est l'élément le plus à droite ou le plus à gauche d'une telle chaîne.
$Lists[i]$ est la tête d'une liste. Chaque élément de cette liste contient un pointeur à un élément de *sort* qui peut être déplacé à la position de l'élément i et un lien à l'élément suivant dans $Lists[i]$. Ces structures sont illustrées à la Fig. 5.4.

NextSort :

1. Faire pointer p à l'élément du début de la liste *prêt*. Mettre à jour *plt* et *prt*.
2. Mettre à jour la liste de l'élément pointé par p, et celle de l'élément pointé par *plt*.
   Mettre à jour s.
3. Enlever s de *Lists[p]*.
4. Enlever l'élément pointé par s de la liste *finie* et celui pointé par p de la liste *prêt*.
5. Déplacer s à p.
6. Ajouter l'élément pointé par p à la liste *finie*, à la gauche de son voisin de droite.
7. **si** (p est *fini*)
8. Mettre à jour le drapeau de l'élément pointé par *plt*,
   et mettre le drapeau pointé par s à *non fermé*.
9. Ajouter l'élément pointé par s à la liste *finie* à la gauche de l'élément pointé par p.
10. **si** l'élément pointé par *plt* est dans m et sa liste est vide.
11. Déplacer la chaîne de l'élément pointé par *plt* de la liste *prêt*
    à la liste *finie* à la gauche de l'élément pointé par p.
    fin si
12. **sinon**
13. Ajouter l'élément pointé par s au début de la liste *prêt*.
14. Transférer la liste de l'élément pointé par p à la liste de l'élément pointé par s
    et mettre la liste de l'élément pointé par p à vide

    fin si
15. Mettre le drapeau de l'élément pointé par p à *non fermé*.
16. Déplacer l'élément pointé par p et ses successeurs de la liste *finie* au début de la liste *prêt*.
17. Déplacer la première chaîne de la liste *prêt* au début de la liste *finie*.

Fig. 5.2 Algorithme 2 pour générer l'extension suivante.

Initialiser sort, ses drapeaux, Lists et les listes *prête* et *finie*.
tant que (la liste *prête* est non vide)
    NextSort.
fin tant que

Fig. 5.3 Algorithme 3 pour générer toutes les extensions.

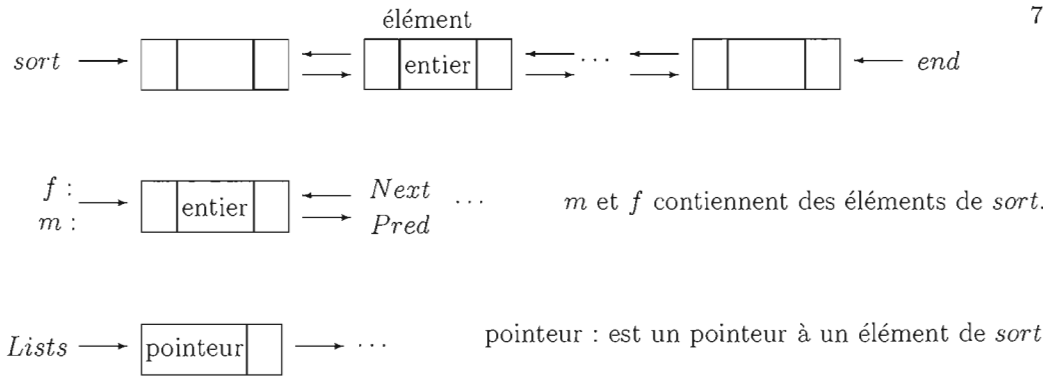

Fig. 5.4 Structures de données utilisées par l'algorithme 2 (Fig. 5.2).

Considérons l'algorithme 2 *NextSort*, Fig. 5.2. Il utilise p pour pointer sur un élément de *sort* auquel nous devons faire le déplacement et, s qui $\in$ *Lists* pour pointer sur l'élément qui doit être déplacé ; *plt* est un pointeur sur l'élément à gauche de l'élément pointé par p, et *prt* sur celui à droite. Grâce aux structures de données choisies, toutes les opérations de l'algorithme requièrent un temps constant (Korsh et Lafollette, 2002), d'où *NextSort* est sans boucle.

5.6 Exemple

Considérons le poset de la Fig. 5.5.

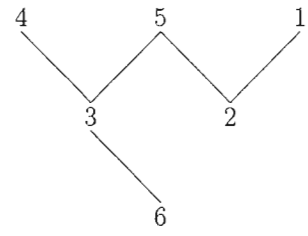

Fig. 5.5 Contraintes $\mathcal{P}$: qu'il y a une ligne de 5 à 3 signifie que $5 < 3$, i.e., *less*[5][3] est vrai.

Les extensions linéaires obtenues par l'algorithme 3 (Fig. 5.3) sont les 26 énumérées à la Fig. 5.6. Dans cette dernière figure, nous voyons les listes associées à chaque élément de *sort* (l'extension linéaire) ainsi que les listes f et m. Aussi, chaque élément de f ou m possède une chaîne dans laquelle nous voyons *Last* et *First* qui sont sûrement correctes pour le dernier et premier éléments d'une chaîne.

L'extension 1) et toutes les autres sont montrées après avoir exécuté *Mettre à jour* à la ligne 2 de *NextSort* (Fig. 5.2).

Les entiers ajoutés aux listes comme résultat de ce *Mettre à jour*, sont montrés en gras. Ce *Mettre à jour*, ajoute 1 dans la liste à la position 3, et 1 dans la liste à la position 6, pour l'extension 1). L'ajout du premier 1 signifie que *Lists*[3] pointe sur son élément qui à son tour contient un pointeur à l'élément 1 de l'extension et un nil puisqu'il n'y a pas d'autres éléments dans *Lists*[3].

Pour l'extension 1), f consiste de 2 et 1 (élément 2 et élément 1) et m consiste de 6, 3, 4 et 5.

Notons que 2 et 1 dans f forment une chaîne de longueur 2 et que *Last*[2]=1, *First*[2]=2, *Last*[1]=1 et *Fist*[1]=2. De façon similaire 6, 3 et 4 forment une chaîne de longueur 3 dans m, alors que 5 forme une chaîne de longueur 1.

Nous allons voir en détail comment passer de l'extension initiale à la suivante.

Pour l'extension initiale, la chaîne la plus à droite est dans la liste *finie*. Toutes les autres chaînes sont dans la liste *prête*. Toutes les listes (*Lists*) sont vides et tous les *drapeaux* sont *non fermés*. Exécutons l'algorithme 2, Fig. 5.2, en commençant avec l'extension :

5 4 3 6 1 2 $\quad$ f : 2(1,2)1(1,2)

$\quad\quad\quad\quad\quad\quad\quad\quad$ m : 6(4,6)3(4,3)4(4,6)5(5,5)

À l'exécution de :

la ligne 1, nous obtenons : $p = 6$, $plt = 3$ et $prt = 1$.

la ligne 2, nous tombons dans les cas

2) de *mise à jour*, page 67, alors nous ajoutons 1 au dessus 6

3) de *mise à jour*, page 67, alors nous ajoutons 1 au dessus 3

nous obtenons alors :

	1 1← s	
	5 4 3 6 1 2	f : 2(1,2)1(1,2)
		m : 6(4,6)3(4,3)4(4,6)5(5,5)
	1	
les lignes 3 et 4,	5 4 3 6 1 2	f : 2(1,2)
		m : 3(4,3)4(4,6)5(5,5)
	1	
la ligne 5,	5 4 3 1 6 2	f : 2(2,2)
		m : 3(4,3)4(4,3)5(5,5)
la ligne 6,		f : 2(2,2)6(6,6)
les lignes 7 et 8,		*flag*[3]=*non fermé* (voir *le mis à jour*, page 67)
		flag[1]=*non fermé*
la ligne 9,		f : 2(2,2)6(6,6)1(1,1)
la ligne 10, rien à faire, car $Lists[3] \neq \emptyset$		
la ligne 15,		*flag*[6]=*non fermé*
la ligne 16,		f : 1(1,1)
		m : 2(2,2)6(6,6)3(4,3)4(4,3)5(5,5)
	1	
la ligne 17 :	5 4 3 1 6 2	f : 2(2,2)1(1,1)
		m : 6(6,6)3(4,3)4(4,3)5(5,5)

De nouveau, nous exécutons *NextSort* :

la ligne 1, nous obtenons : $p = 6$, $plt = 1$ et $prt = 2$.

la ligne 2,	1 **2**← s	
	5 4 3 1 6 2	

...

Pour générer toutes les extensions linéaires, il suffit d'exécuter l'algorithme 3 (Fig. 5.3).

	L'extension initiale ——> 5 4 3 6 1 2	2(1,2)1(1,2) 6(4,6)3(4,3)4(4,6)5(5,5)
	1 1	
1)	5 4 3 6 1 2	2(1,2)1(1,2) 6(4,6)3(4,3)4(4,6)5(5,5)
	1 **2**	
2)	5 4 3 1 6 2	2(2,2)1(1,1) 6(6,6)3(4,3)4(4,3)5(5,5)
	1 1	
3)	5 4 3 1 2 6	6(6,6)2(1,2)1(1,2) 3(4,3)4(4,3)5(5,5)
	1 **2 6**	
4)	5 4 1 3 2 6	6(6,6)1(1,1) 2(2,2)3(3,3)4(4,4)5(5,5)
	1 2	
5)	5 4 1 3 6 2	2(2,2)6(6,6)1(1,1) 3(3,3)4(4,4)5(5,5)
	4	
	1 1	
6)	5 4 1 2 3 6	6(3,6)3(3,6)2(1,2)1(1,1) 4(4,4)5(5,5)
	4	
	1 **2 3**	
7)	5 1 4 2 3 6	6(3,6)3(3,6)1(1,1) 2(2,2)4(4,4)5(5,5)
	4	
	1 2 **6**	
8)	5 1 4 3 2 6	6(6,6)3(3,3)1(1,1) 2(2,2)4(4,4)5(5,5)
	4	
	1 2	
9)	5 1 4 3 6 2	2(2,2)6(3,6)3(3,6)1(1,1) 4(4,4)5(5,5)
	4	
	1	
10)	5 1 2 4 3 6	6(4,6)3(3,6)4(4,6)2(1,2)1(1,2) 5(5,5)
	4 **4 4**	
11)	1 5 2 4 3 6	6(4,6)3(3,6)4(4,6) 2(5,2)5(5,2)1(1,1)
	4 4 **3**	
12)	1 5 4 2 3 6	6(3,6)3(3,6)4(4,4) 2(2,2)5(5,5)1(1,1)
	4 4 **6**	
13)	1 5 4 3 2 6	6(6,6)3(4,3)4(4,3) 2(2,2)5(5,5)1(1,1)
	4 4	
14)	1 5 4 3 6 2	2(2,2)6(4,6)3(4,3)4(4,6) 5(5,5)1(1,1)
	4 **2 2**	
15)	1 4 5 3 6 2	2(2,2)4(4,4) 6(5,6)3(3,6)5(5,6)1(1,1)
	4 2	
16)	1 4 5 3 2 6	6(6,6)2(2,2)4(4,4) 3(5,6)5(5,3)1(1,1)
	4	
17)	1 4 5 2 3 6	6(3,6)3(3,6)2(5,2)5(5,2)4(4,4) 1(1,1)
	3	
18)	4 1 5 2 3 6	6(3,6)3(3,6)4(4,4) 2(5,2)5(5,2)1(1,1)
	6	
19)	4 1 5 3 2 6	6(6,6)3(5,3)5(5,3)4(4,4) 2(2,2)1(1,1)
	5	
20)	4 1 5 3 6 2	2(2,2)6(5,6)3(5,3)5(5,6)4(4,4) 1(1,1)
	2 2	
21)	4 5 1 3 6 2	2(2,2)5(5,5)4(4,4) 6(3,6)3(3,6)1(1,1)
	3 2	
22)	4 5 1 3 2 6	6(6,6)2(2,2)5(5,5)4(4,4) 3(3,3)1(1,1)
	3	
23)	4 5 1 2 3 6	6(3,6)3(3,6)2(2,2)5(5,5)4(4,4) 1(1,1)
	6 6	
24)	4 5 3 1 2 6	6(6,6)3(5,3)5(5,3)4(4,4) 2(1,2)1(1,2)
	6	
25)	4 5 3 1 6 2	2(2,2)6(6,6)3(5,3)5(5,3)4(4,4) 1(1,1)
26)	4 5 3 6 1 2	2(1,2)1(1,2)6(5,6)3(5,3)5(5,6)4(4,4)

Fig. 5.6 Extensions linéaires obtenues par l'algorithme 3 (Fig. 5.3) pour le poset de la Fig. 5.5.

CHAPITRE VI

DEUX CODES GRAY 2-CLOSE POUR LES COMBINAISONS AVEC IMPLANTATION SANS BOUCLE

6.1 Introduction

Vincent Vajnovszki et Timothy Walsh ont montré dans un rapport de recherche (Vajnovszki et Walsh, 2003) la similarité de deux codes Gray pour les combinaisons (chaînes binaires de longueur n avec m occurrences de 1) qui sont présentés séparément par P. Chase (Chase, 89) et F. Ruskey (Ruskey, 93). Un code Gray 2-close (voir la définition 6.5) pour les combinaisons a été découvert par P. Chase (Chase, 89). Il a donné une description obscure non-récursive de son 1-code et un programme FORTRAN pour une implantation sans boucle de son 0-code (voir la définition 6.2). Ensuite, F. Ruskey (Ruskey, 93) a publié une description récursive d'un autre code Gray 2-close pour les combinaisons.

V. Vajnovszki et T. Walsh (Vajnovszki et Walsh, 2003) ont donné une description plus simple et une implantation sans boucle du 0-code de Chase (Chase, 89) et une description non-récursive du code Gray de Ruskey (Ruskey, 93).

6.2 Définitions

Définition 6.1 *Une* ***combinaison*** *de m éléments pris parmi n éléments d'un ensemble E à n éléments est un sous ensemble de m éléments pris parmi les n éléments de E. Nous utiliserons dans ce cas la notation m-combinaison.*

Remarque 6.1 *Une m-combinaison de n objets peut être codée par une chaîne binaire w avec m 1 et $n - m$ 0.*

Définition 6.2 *Soit w une chaîne binaire. Nous pouvons utiliser un vecteur de longueur m dont la i-ième composante est la position de la i-ième occurrence de 1 dans w. Une telle représentation est appelée* ***1-code*** *de w et nous définissons de façon similaire le* ***0-code****.*

Définition 6.3 *Un code Gray pour les combinaisons est appelé* ***minimal*** *si toute chaîne binaire peut être transformée en la suivante en échangeant un seul 1 avec un 0, ce qui implique que dans son 1-code ou 0-code correspondant, au plus deux chiffres changent d'un mot au suivant.*

Il y a, dans la littérature, beaucoup de codes Gray minimaux pour les combinaisons. Le plus simple est celui de C. N. Liu et D. T. Tang (Liu et Tang, 73) (voir le Tab. 6.1) dont une implantation sans boucle a été donné par T. Walsh (Walsh, 95). Ehrlich (Ehrlich, 73) a aussi découvert un autre code Gray pour les combinaisons avec implantation sans boucle.

Définition 6.4 *Un code Gray pour les combinaisons est appelé* ***homogène*** *si les chiffres 1 et 0 qui s'échangent sont séparés seulement par des 0, ce qui signifie que dans le 1-code, un seul chiffre change de valeur d'un mot au suivant.*

Un tel code a été découvert par P. Eades et B. McKay (Eades et McKay, 84) (voir le Tab. 6.1). Une description non-récursive et une implantation sans boucle ont été données par T. Walsh (Walsh, 2001). Nous verrons cette dernière au chapitre 8.

Définition 6.5 *Un code Gray homogène est appelé* ***2-close*** *si le 1 et le 0 qui s'échangent sont, ou bien adjacents, ou bien séparés par un seul 0, ce qui signifie que dans le 1-code, le chiffre qui change de valeur le fait par au plus deux.*

Liu-Tang		Eades-McKay		Chase			Ruskey	
111000	123	111000	123	000111	456	123	001110	345
101100	134	110100	124	010011	256	134	100110	145
011100	234	101100	134	100011	156	234	010110	245
110100	124	011100	234	001011	356	124	011010	235
100110	145	011010	235	001101	346	125	101010	135
010110	245	101010	135	010101	246	135	110010	125
001110	345	110010	125	100101	146	235	111000	123
101010	135	100110	145	110001	126	345	110100	124
011010	235	010110	245	101001	136	245	101100	134
110010	125	001110	345	011001	236	145	011100	234
100011	156	001101	346	011100	234	156	011001	236
010011	256	100101	146	101100	134	256	101001	136
001011	356	010101	246	110100	124	356	110001	126
000111	456	011001	236	111000	123	456	100101	146
100101	146	101001	136	110010	125	346	010101	246
010101	246	110001	126	101010	135	246	001101	346
001101	346	100011	156	011010	235	146	001011	356
101001	136	010011	256	010110	245	136	100011	156
011001	236	001011	356	100110	145	236	010011	256
110001	126	000111	456	001110	345	126	000111	456

Tab. 6.1 Codes Gray de Liu-Tang, Eades-McKay et Chase et Ruskey, pour les 3-combinaisons de $\{1, 2, 3, 4, 5, 6\}$ en chaînes binaires et en 1-code, pour celui de Chase le 0-code est montré aussi.

Définition 6.6 *Nous disons qu'une liste de mots est* ***en partition préfixe*** *(respectivement* ***suffixe****) si tous les mots dans la liste avec le même préfixe (respectivement suffixe) forment un intervalle de mots adjacents.*
Dans l'intervalle dans lequel un préfixe de longueur $i-1$ reste constant (si la liste est en partition préfixe), le i-ième chiffre prend une suite de valeurs distinctes déterminées par le préfixe ; cette suite, en tant que fonction du préfixe, définit le premier et le dernier mots de la liste et le successeur de chaque mot, ce qui constitue une description non-récursive de la liste de mots.

Remarque 6.2 *Dans la littérature, la plupart de codes Gray sont en partition préfixe ou suffixe ou encore, peuvent être transformés en de telles listes.*

Définition 6.7 *Nous appelons* ***pivot****, dans une liste de mots en partition préfixe, l'indice du chiffre le plus à gauche qui change de valeur en passant d'un mot au suivant ; dans une liste en partition suffixe, c'est le chiffre le plus à droite.*

6.3 Code Gray 2-close de Chase

Soit $(c[1], c[2], \cdots, c[m])$, le 0-code d'une $(n-m)$-combinaison de n. Comme $c[i]$ est la position de la i-ième occurrence de 0 dans la chaîne binaire, alors $1 \leq c[1] < c[2] < \cdots < c[m] \leq n$. En exécutant le programme FORTRAN (Fig. 6.1) de Chase (Chase, 89), V. Vajnovszki et T. Walsh ont fait l'observation suivante :

Théorème 6.1 *Le code Gray pour les combinaisons en représentation 0-code généré par le programme FORTRAN de Chase (Chase, 89) est en partition suffixe avec la description suivante : $c[m]$ augmente par 1 de sa valeur minimale m à sa valeur maximale n, et pour chaque i de $m-1$ à 1 dans chaque intervalle de mots avec le même suffixe $(c[i+1], \cdots, c[m])$, si $c[i+1]$ est impair, alors $c[i]$ augmente par 1 de sa valeur minimale i à sa valeur maximale $c[i+1]-1$, et sinon, diminue par 1 de sa valeur maximale à sa valeur minimale.*

Preuve : La démonstration suivante nous permet de passer par tous les cas possibles de l'algorithme de Chase (dont le programme FORTRAN se trouve à la Fig. 6.1 et une version plus structurée est écrite à la Fig. 6.2) et d'exhiber tous les cas possibles du théorème.

Notons que nous nous intéressons seulement aux valeurs changées dans le tableau c de dimension $l > m$, et à la valeur z qui est l'indice minimal j pour lequel $c[j] > j$. De plus, nous écrivons seulement les valeurs finales qui nous intéressent.

Les remarques 6.3, 6.4 et 6.5 prouvent que certaines affectations fausses de z dans l'algorithme de Chase (Fig 6.2) sont corrigées à l'intérieur de l'algorithme afin de donner des résultats vrais.

1. ***c[1]=1, c[2] est pair et c[z] est impair*** (*les lignes* $22 \rightarrow 26$).

   (a) ***Si*** z ***est impair*** : Dans ce cas $c[z-1] \leftarrow z$ et $z \leftarrow z-1$ (selon Chase).

   Selon le théorème 6.1 : $c[z]$ est impair et $z \geq 2$. Alors $c[z-1]$ augmente et comme z est impair, alors $c[z-1] = z-1$ qui est pair, d'où $c[z-1] < c[z]-1$. Par conséquent, $c[z-1]$ peut augmenter et nous obtenons $c[z-1] \leftarrow z$. Maintenant $c[z-1]$ est impair; alors $c[z-2]$ ne change pas, car il doit commencer de $z-2$ qui est sa valeur actuelle, et z devient $z-1$.

   (b) ***Si*** z ***est pair*** : Dans ce cas, $c[z-1] \leftarrow z$, $z \leftarrow z-2$ et $c[z-2] \leftarrow z-1$ (Chase).

   Selon le théorème 6.1 : $c[z]$ est impair et $z \geq 2$. Alors $c[z-1]$ augmente et comme z est pair, alors $c[z-1] = z-1$ qui est impair, d'où $c[z-1] < c[z]+1$. Par conséquent, $c[z-1]$ peut augmenter et nous obtenons $c[z-1] \leftarrow z$. Maintenant $c[z-1]$ est pair, alors $c[z-2]$ diminue à partir de $c[z-1]-1 = z-1$. Finalement $c[z-2]$ est impair et il n'y a pas de changement à sa gauche et $c[z-2] > z-2$, d'où $z \leftarrow z-2$.

2. ***c[1]*** $\neq$ ***1 et*** $c[2]$ ***est pair*** (*les lignes* $29 \rightarrow 32$).

   $c[1] \leftarrow c[1]-1$ et $z \leftarrow 2$ (Chase) (Cette affectation est fausse si $c[1] \neq 1$; voir la remarque 6.3).

Selon le théorème 6.1 : $c[2]$ est pair et $c[1] > 1$, alors $c[1]$ peut diminuer à $c[1] - 1$ et z devient 2 si $c[1] = 1$, sinon z devrait être 1.

3. ***c[2] est impair.***

   (a) ***Si c[2] > c[1]+1*** (*les lignes* $43 \rightarrow 45$) : alors $c[1] \leftarrow c[1] + 1$ (Chase) (voir la remarque 6.5).

   Selon le théorème 6.1 : $c[2]$ est impair et le chiffre à sa gauche $c[1] < c[2] - 1$, alors $c[1]$ peut augmenter, d'où $c[1] \leftarrow c[1] + 1$.

   (b) ***Si c[2] = c[1]+1*** (*les lignes* $36 \rightarrow 41$) :

      i. ***Si c[3] est pair*** : alors $z = 3$ (qui n'est vraie que si $c[1] = 2$; voir la remarque 6.4), $c[1] \leftarrow c[1] - 1$ et $c[2] \leftarrow c[1] + 1$ (Chase).

      Selon le théorème 6.1 : $c[2]$ est impair et $c[2] = c[1] + 1$. Alors $c[1]$ est pair (i.e. $z = 1$) et ne peut pas augmenter, donc nous regardons $c[z+1]$, i.e. $c[3]$ qui est pair. Alors, $c[2]$ peut diminuer pour devenir pair et dans ce cas-là, $c[1]$ diminue (à partir de sa valeur maximale) :

      $c[2] \leftarrow c[2] - 1$ et $c[1] \leftarrow c[2] - 1$ qui sont équivalents à : $c[1] \leftarrow c[1] - 1$ et $c[2] \leftarrow c[1] + 1$. Et si $c[1] = 1$, alors $z = 3$.

      ii. ***Si c[3] est impair*** : $z = 3$ (qui n'est pas vraie), $c[1] \leftarrow c[1] + 1$ et $c[2] \leftarrow c[1] + 1$ (Chase).

      Selon le théorème 6.1 : $c[2]$ est impair et $c[2] = c[1] + 1$. Alors $c[1]$ est pair (i.e. $z = 1$) et ne peut pas augmenter, donc nous regardons $c[z+1]$, i.e. $c[3]$ qui est impair, et $c[3] > c[2] + 1$ (car $c[2]$ est aussi impair). Alors, $c[2]$ peut augmenter pour devenir pair et dans ce cas-là, $c[1]$ diminue (à partir de sa valeur maximale) :

      $c[2] \leftarrow c[2] + 1$ et $c[1] \leftarrow c[2] - 1$ qui sont équivalents à : $c[1] \leftarrow c[1] + 1$ et $c[2] \leftarrow c[1] + 1$.

      Et z ne change pas.

4. ***c[1]=1, c[2] est pair et c[z] est pair.***

   (a) ***Si*** $c[z+1] > c[z] + 1$ (*les lignes* $14 \rightarrow 19$) :

i. ***Si*** $c[z+1]$ ***est impair*** : $c[z] \leftarrow c[z+1]$ et comme $c[z] \neq z$ alors z ne change pas (Chase).

Selon le théorème 6.1 : $c[z]$ est pair et $c[z+1]$ est impair, alors $c[z]$ va augmenter et cela est possible, car $c[z+1] > c[z]+1$, d'où $c[z] \leftarrow c[z]+1$; $c[z]$ devient impair et $c[z-1] = z-1$ qui est à sa première valeur, ne change pas, d'où z ne change pas non plus.

ii. ***Si*** $c[z+1]$ ***est pair*** : $c[z] \leftarrow c[z]-1$ et ici z peut changer à $z+1$ si $c[z] = z$ (Chase).

Selon le théorème 6.1 : $c[z]$ est pair et $c[z+1]$ est pair, alors $c[z]$ diminue, d'où $c[z] \leftarrow c[z]-1$; si $c[z] = z$, alors z devient $z+1$, car $c[z+1] > z+1$.

(b) ***Si*** $c[z+1] = c[z]+1$ (*les lignes* $6 \rightarrow 12$) :

i. **Si** $c[z+2]$ **est pair** : $c[z+1] \leftarrow c[z+1]-1$, $c[z] \leftarrow c[z]-1$ et si $c[z] = z$, alors $z \leftarrow z+2$ (Chase).

Selon le théorème 6.1 : $c[z]$ est pair (aucun chiffre à sa droite ne change) et $c[z+1] = c[z]+1$. Alors, $c[z+1]$ est impair et $c[z]$ ne peut pas augmenter, donc nous regardons $c[z+2]$ qui est pair. Alors, $c[z+1]$ diminue (et c'est possible) et devient $c[z+1] \leftarrow c[z+1]-1$ qui est pair, d'où $c[z]$ commence à diminuer aussi à partir de $c[z+1]-1$ qui est maintenant égale à $c[z]-1$; si $c[z] = z$, alors $z \leftarrow z+2$.

ii. ***Si*** $c[z+2]$ ***est impair*** : $c[z+1] \leftarrow c[z+1]+1$, $c[z] \leftarrow c[z]+1$ et comme $c[z] \neq z$, alors z ne change pas (Chase).

Selon le théorème 6.1 : $c[z]$ est pair (aucun chiffre à sa droite ne change) et $c[z+1] = c[z]+1$, alors $c[z+1]$ est impair et $c[z]$ ne peut pas augmenter, donc nous regardons $c[z+2]$ qui est impair. Alors, $c[z+1]$ augmente (et c'est possible, car $c[z+1] < c[z+2]-1$) et devient $c[z+1] \leftarrow c[z+1]+1$ qui est pair, d'où $c[z]$ commence à diminuer aussi à partir de $c[z+1]-1$ qui est maintenant égale à $c[z]+1$ et z ne change pas. □

V. Vajnovszki et T. Walsh ont utilisé l'astuce de Chase : ils ont gardé une variable z, la valeur minimale de i telle que $c[i] > i$. Ce qui fait que l'implantation est sans boucle,

c'est que le pivot doit être $z-1$ ou z ou $z+1$.

Leur algorithme est à la Fig. 6.3. Ils ont pris $(1, 2, \cdots, m)$ comme valeur initiale de $(c[1], c[2], \cdots, c[m])$ et pour éviter le test de la fin du tableau, ils ont mis $c[m+1] = 2n+1$, (impair, ce qui oblige l'incrémentation de $c[m]$). Nous arrêtons l'exécution de cette procédure quand $c[m]$ dépasse n.

Remarque 6.3 *Dans le cas où c[2] est pair et c[1]≠ 1 (les lignes 29 → 32), en appliquant l'algorithme de Chase (Fig. 6.2) successivement, c[1] diminue jusqu'à 1, d'où l'affectation z=2 dans chacune des itérations. Cette affectation est fausse, sauf pour la dernière, mais n'a pas d'effet erroné.*

Par exemple, pour m=3 et n=7 et en appliquant l'algorithme de Chase (Fig. 6.2) à la combinaison 3 4 6 15, nous obtenons :

3 4 6 15

2 4 6 15

1 4 6 15.

Remarque 6.4 *Dans le cas où c[2] est impair, c[1]=c[2]-1 et c[3] est pair (les lignes 36 → 41) ; c[2] diminue à c[2]-1 et devient pair, alors c[1] diminue aussi à c[1]-1 et là :*

- *soit nous nous ramenons à la remarque 6.3. Par exemple, pour m=3 et n=7 et en appliquant l'algorithme de Chase (Fig. 6.2) à la combinaison 4 5 6 15, nous obtenons :*

  4 5 6 15

  3 4 6 15

- *soit nous tombons dans le cas où c[1]=1, c[2]=2 et c[3] > 3. Par exemple, pour m=3 et n=7 et en appliquant l'algorithme de Chase (Fig. 6.2) à la combinaison 2 3 6 15, nous obtenons :*

  2 3 6 15

  1 2 6 15.

Dans le cas où c[2] est impair, c[1]=c[2]-1 et c[3] est impair (les lignes 36 → 41) ; c[2] augmente à c[2]+1 et devient pair, alors c[1] augmente aussi à c[1]+1 et là nous

nous ramenons à la remarque 6.3. Par exemple, pour m=3 et n=7 et en appliquant l'algorithme de Chase (Fig. 6.2) à la combinaison 2 3 5 15, nous obtenons :

2 3 5 15

3 4 5 15,

d'où l'affectation z=3 ne fait rien.

Remarque 6.5 *Dans le cas où c[2] est impair et c[2] > c[1]+1 ; c[1] augmente jusqu'à c[2]-1 et là, nous nous ramenons à la remarque 6.4. Par exemple, pour m=3 et n=7 et en appliquant l'algorithme de Chase (Fig. 6.2) à la combinaison 1 5 6 15, nous obtenons :*

1 5 6 15

2 5 6 15

3 5 6 15

4 5 6 15,

d'où le fait de ne pas changer la valeur de z ne fait rien, car d'abord nous nous ramenons à la remarque 6.4 qui, à son tour, nous ramène à la remarque 6.3 qui donne la bonne valeur de z à la dernière itération.

6.3.1 Exemple

Pour $n = 6$ et $m = 3$, exécutons l'algorithme à la Fig. 6.3 en commençant par c : 1 2 3 13, qui sans 13 correspond en 0-code à la chaîne binaire : 0 0 0 1 1 1 . La trace se trouve à la Fig. 6.4. Nous voyons comment la variable z (indiquée par un flèche) change pour le tableau c (sur la partie gauche de la Fig. 6.4), l'élément de la chaîne binaire dont c est le 0-code et la position à laquelle il se déplace (sur la partie droite de la Fig. 6.4).

Remarque. Ce que nous obtenons, ce sont les 3-combinaisons d'un ensemble de 6 éléments. Nous voyons clairement que la liste des chaînes binaires est homogène et 2-close. Ainsi, dans le tableau c, il y a au plus 2 chiffres qui changent de valeurs en passant d'une itération à l'autre (d'un mot à l'autre).

```
      SUBROUTINE TWID ( C, X, Y, Z, L )
      IMPICIT INTEGER (A-Z)
      DIMENSION C(L)

      IF (MOD(C(2),2).EQ.0) GO TO 50
      IF (C(2).EQ.C(1)+1) GO TO 10
      X=C(1)
      Y=C(1)+1
      C(1)=Y
      RETURN

10    I=MOD(C(3),2)
      Z=3
      C(1)=C(1)+I+I-1
      X=C(2)-I
      C(2)=C(1)+1
      Y=C(1)+I
      RETURN

20    IF (MOD(C(Z),2).EQ.0) GO TO 30
      Y=Z
      X=Z+MOD(Z,2)-2
      C(Z-1)=Z
      Z=X
      C(X)=X+1
      RETURN

30    IF (C(Z+1).EQ.C(Z)+1) GO TO 40
      J=MOD(C(Z+1),2)
      J=J+J-1
      X=C(Z)
      Y=X+J
      C(Z)=Y
* NEXT STATEMENT'S EFFECT : IF(Y.EQ.Z) Z=X
      Z=Z+IDIM(Z,Y-1)
      RETURN

40    I=MOD(C(Z+2),2)
      J=3*I-1
      X=C(Z+1)-I
      Y=C(Z)+J
      C(Z+1)=X+J
      C(Z)=Y-I
* NEXT STATEMENT'S EFFECT : IF(Y.EQ.Z) Z=X
      Z=Z+2*IDIM(Z,Y-1)
      RETURN

50    IF ((C(1).EQ.1) GO TO 20
      X=C(1)
      Z=2
      Y=X-1
      C(1)=Y
      RETURN

      END
```

Fig. 6.1 Algorithme de Chase en FORTRAN

```
1.  NEXT (c, x, y, z, l)
2.      if (c[2] is even) then
3.          if (c[1] = 1) then
4.              if (c[z] is even) then
5.                  if (c[z + 1] = c[z] + 1) then
6.                      i ← c[z + 2] mod2 ;
7.                      j ← 3i − 1 ;
8.                      x ← c[z + 1] − i ;
9.                      y ← c[z] + j ;
10.                     c[z + 1] ← x + j ;
11.                     c[z] ← y − i ;
12.                     if (y = z) then z ← x end if
13.                 else {c[z + 1] > c[z] + 1}
14.                     j ← c[z + 1] mod2 ;
15.                     j ← 2j − 1 ;
16.                     x ← c[z] ;
17.                     y ← x + j ;
18.                     c[z] ← y ;
19.                     if (y = z) then z ← x end if
20.                 end if
21.             else {c[z] is odd}
22.                 y ← z ;
23.                 x ← z + (z mod2) − 2 ;
24.                 c[z − 1] ← z ;
25.                 z ← x ;
26.                 c[x] ← x + 1 ;
27.             end if
28.         else {c[1] > 1}
29.             x ← c[1] ;
30.             z ← 2 ;
31.             y ← x − 1 ;
32.             c[1] ← y
33.         end if
34.     else {c[2] is odd}
35.         if (c[2] = c[1] + 1) then
36.             i ← c[3] mod2 ;
37.             z ← 3 ;
38.             c[1] ← c[1] + 2i − 1 ;
39.             x ← c[2] − i ;
40.             c[2] ← c[1] + 1 ;
41.             y ← c[1] + i ;
42.         else {c[2] > c[1] + 1}
43.             x ← c[1] ;
44.             y ← c[1] + 1 ;
45.             c[1] ← y ;
46.         end if
47.     end if
48. end NEXT.
```

Fig. 6.2 Algorithme de Chase. Initialement $c[i] = i$ pour $1 \leq i \leq m$, $c[m+1] = 2n+1$, $z = m+1$, $l =$ dimension de c qui est $> m$, x est l'élément enlevé et y celui ajouté.

```
Procedure Next
   if (z > 1) and (c[z] is odd) then          {c[z-1] existe et peut augmenter, car}
                                                {c[z-1]=z-1 et c[z]>z}
      c[z − 1] := c[z − 1] + 1 ;
      if (c[z − 1] is even) and (z > 2) then   {c[z-2] existe et commence à sa}
                                                {valeur maximale}
         c[z − 2] := c[z − 1] − 1
         z := z − 2        {car c[z-1] est maintenant z et c[z-2] est maintenant z-1}
      else
         z := z − 1
      end if
   else        {soit c[z-1] n'existe pas, soit il ne peut pas bouger ; nous essayons c[z]}
      if c[z + 1] is even {c[z]>z ; alors il peut diminuer} then
         c[z] := c[z] − 1
         if c[z] = z then z := z + 1 end if
      else
         if c[z] < c[z + 1] − 1 then                          {c[z] peut augmenter}
            c[z] := c[z] + 1
                  {dans les deux cas, c[z] est maintenant impair ; d'où si z>1,}
                                                {c[z-1] reste à sa valeur minimale}
         else                                 {c[z+1] est impair et c[z]=c[z+1]-1 ; alors}
                                  {c[z] ne peut pas bouger et nous essayons c[z+1]}
            if (c[z + 2] is odd) then          {c[z+1]<c[z+2]-1 ; alors c[z+1]}
                                                {peut augmenter}
               c[z + 1] := c[z + 1] + 1 ;
            else          {c[z+2] est pair et c[z+1]>z+1 ; alors c[z+1] diminue}
               c[z + 1] := c[z + 1] − 1 ;
            end if ;
            c[z] := c[z + 1] − 1            {c[z+1] est maintenant pair ; alors c[z]}
                                                {commence à sa valeur maximale}
            if c[z] = z then z := z + 2 end if          {puisque c[z+1]=z+1}
         end if
      end if
   end if
end Next.
```

Fig. 6.3 Code Gray sans boucle pour les combinaisons.

Le tableau c	La chaîne binaire dont c est le 0-code
1 2 3 13	0 0 0 1 1 1
1 3 4	0 1 0 0 1 1
2 3 4	1 0 0 0 1 1
1 2 4	0 0 1 0 1 1
1 2 5	0 0 1 1 0 1
1 3 5	0 1 0 1 0 1
2 3 5	1 0 0 1 0 1
3 4 5	1 1 0 0 0 1
2 4 5	1 0 1 0 0 1
1 4 5	0 1 1 0 0 1
1 5 6	0 1 1 1 0 0
2 5 6	1 0 1 1 0 0
3 5 6	1 1 0 1 0 0
4 5 6	1 1 1 0 0 0
3 4 6	1 1 0 0 1 0
2 4 6	1 0 1 0 1 0
1 4 6	0 1 1 0 1 0
1 3 6	0 1 0 1 1 0
2 3 6	1 0 0 1 1 0
1 2 6	0 0 1 1 1 0
7	

Fig. 6.4 Trace de l'algorithme (Fig. 6.3) pour $n = 6$ et $m = 3$.

6.4 Code Gray 2-close de Ruskey

Notations. Nous notons 0^m (respectivement 1^m), la chaîne constituée de m 0 consécutifs (respectivement m 1 consécutifs). Nous notons $L1$ (où L est un langage), la liste de tous les mots de L suivis de 1 et de façon similaire, nous notons $L0$ (respectivement $0L$), la liste de tous les mots de L suivis de 0 (respectivement précédés de 0). Nous notons L^R, la liste L renversée (lu du dernier mot jusqu'au premier) et la liste L suivie par la liste M est notée L, M.

Le code Gray de Ruskey utilise deux listes ; V. Vajnovszki et T. Walsh (Vajnovszki et Walsh, 2003) ont modifié sa description récursive de telle sorte qu'une seule liste soit nécessaire.

Soit $L(n,m)$, la liste des chaînes binaires avec m 1 et $n-m$ 0. Alors le code Gray de Ruskey peut être décrit récursivement comme suit :

$$
\begin{aligned}
L(n,n) &= 1^n \\
L(n,0) &= 0^n \\
L(n,n-1) &= 1^{n-1}0, L(n-1,n-2)1 \qquad (= 1^{n-1}0, 1^{n-2}01, \cdots, 01^{n-1}) \\
L(n,1) &= L^R(n-1,1)0, 0^{n-1}1 \\
L(n,m) &= L^R(n-1,m)0, L(n-2,m-1)01, L(n-2,m-2)11 \text{ if } 1 < m < n-1.
\end{aligned}
$$

Notons que le premier mot dans $L(n,m)$ est $0^{n-m-1}1^m0$ et le dernier est $0^{n-m}1^m$.

6.4.1 Exemple

Trouvons à partir de cette description $L(6,3)$, qui est l'équivalent du code Gray de Chase de l'exemple 6.3.1, à la page 82.

$$
L(2,1) = \begin{cases} L^R(1,1)\ 0 \\ 0\ 1 \end{cases} = \begin{cases} 1\ 0 \\ 0\ 1 \end{cases}
$$

$$
L(3,1) = \begin{cases} L^R(2,1)\ 0 \\ 0\ 0\ 1 \end{cases} = \begin{cases} 0\ 1\ 0 \\ 1\ 0\ 0 \\ 0\ 0\ 1 \end{cases}
$$

$L(3,2) = L^R(2,2)\ 0\ ,\ L(1,1)\ 0\ 1\ ,\ L(1,0)\ 1\ 1$

1 1 0 1 0 1 0 1 1

$L(4,2) = L^R(3,2)\ 0\ ,\ L(2,1)\ 0\ 1\ ,\ L(2,0)\ 1\ 1$

0 1 1 0 1 0 0 1 0 0 1 1

1 0 1 0 0 1 0 1 0 1

1 1 0 0

$L(4,3) = 1\ 1\ 1\ 0\ ,\ L(3,2)\ 1$

$L(4,3) = 1\ 1\ 1\ 0\ ,\ 1\ 1\ 0\ 1\ ,\ 1\ 0\ 1\ 1\ ,\ 0\ 1\ 1\ 1$

$L(5,3) = L^R(4,3)\ 0\ ,\ L(3,2)\ 0\ 1\ ,\ L(3,1)\ 1\ 1$

0 1 1 1 0 1 1 0 0 1 0 1 0 1 1

1 0 1 1 0 1 0 1 0 1 1 0 0 1 1

1 1 0 1 0 0 1 1 0 1 0 0 1 1 1

1 1 1 0 0

$L(6,3) = L^R(5,3)\ 0\ ,\ L(4,2)\ 0\ 1\ ,\ L(4,1)\ 1\ 1$

0 0 1 1 1 0 0 1 1 0 0 1 0 0 1 0 1 1

1 0 0 1 1 0 1 0 1 0 0 1 1 0 0 0 1 1

0 1 0 1 1 0 1 1 0 0 0 1 0 1 0 0 1 1

0 1 1 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1

1 0 1 0 1 0 0 1 0 1 0 1

1 1 0 0 1 0 0 0 1 1 0 1

1 1 1 0 0 0

1 1 0 1 0 0

1 0 1 1 0 0

0 1 1 1 0 0

Nous obtenons la même liste de chaînes binaires de l'exemple 6.3.1, mais renversée dans ce cas-là (comparer avec la partie droite de la Fig. 6.4).

Théorème 6.2 *Le 0-code du code Gray défini à la page 87 est en partition suffixe avec la description suivante : $c[n-m]$ diminue par 1 de sa valeur maximale n à sa valeur minimale $n-m$, et pour chaque i de $n-m-1$ à 1 dans chaque intervalle de mots avec le même suffixe $(c[i+1], \cdots, c[n-m])$, si $n - c[i+1]$ est pair, alors $c[i]$ augmente par 1 de sa valeur minimale i à sa valeur maximale $c[i+1]-1$, et sinon, diminue par 1 de sa valeur maximale à sa valeur minimale.*

Preuve : voir (Vajnovszki et Walsh, 2003).

En comparant les théorèmes 6.1 et 6.2, nous pouvons voir que le code Gray 2-close de Ruskey pour les combinaisons est identique à celui de Chase pour n pair, sauf que la liste est lue à partir du dernier mot jusqu'au permier ; pour n impair, les sous-listes de mots avec $c[m]$ fixe sont les mêmes dans les deux codes Gray, mais leur ordre est renversé.

CHAPITRE VII

CODE GRAY DE PROSKUROWSKI ET RUSKEY POUR LES CHAÎNES DE PARENTHÈSES BIEN FORMÉES ET UNE DESCRIPTION NON-RÉCURSIVE

7.1 Introduction

Timothy Walsh (Walsh, 98) a trouvé le premier algorithme sans boucle pour la génération de chaînes de parenthèses bien formées ; il l'a conçu à partir de l'algorithme récursif de Proskurowski et Ruskey (Proskurowski et Ruskey, 90), qui ont défié le lecteur à trouver une version sans boucle.

En fait, T. Walsh a trouvé deux algorithmes. Dans l'un, il a appliqué l'ordre *graylex* (voir la définition 7.4) de Chase (Chase, 89) pour dériver une version non-récursive qui génère chaque chaîne dans le pire des cas en temps $O(n)$ et utilise un espace auxiliaire de $O(1)$. Dans l'autre, il a appliqué le tableau auxiliaire d'Ehrlich (Bitner, Ehrlich et Reingold, 76; Ehrlich, 73) pour dériver une version non-récursive qui génère chaque chaîne dans le pire des cas en temps $O(1)$ et utilise un espace auxiliaire de $O(n)$.

Nous allons présenter le premier algorithme.

7.2 Terminologie et définitions

Définition 7.1 *Un **mot de Dyck**, ou une chaîne de parenthèses bien formée de longueur $2n$, est une chaîne de n 0 et n 1 telle qu'aucun préfixe ne contient plus de 0 que de 1.*

Définition 7.2 *Pour le mot de Dyck $y = 1^k0x$, nous définissons les fonctions* ***flip*** *et* ***insert*** *par :*

$flip(y) = 1^{k-1}01x$ *et* $insert(y) = 1^{k+1}00x$.

Lorsque le paramètre de flip et/ou insert est une liste, nous appliquons l'une et/ou l'autre fonction à chaque mot de la liste.

Remarque 7.1 *Rappelons que A^R désigne la liste A renversée, et notons AoB, la liste A suivie par la liste B.*

7.3 Code Gray de Proskurowski et Ruskey

Proskurowski et Ruskey (Proskurowski et Ruskey, 90) ont présenté la description récursive suivante d'un code Gray dans lequel chaque mot de longueur $2n$ est changé en son successeur par une transposition de 2 lettres. Notons $T(n,k)$, $1 \leq k \leq n$, les listes constituées de mots de longueur $2n$ possédant 1^k0 comme préfixe et qui sont générées comme suit :

$$T(n,k) = \begin{cases} flip(T(n,2)) & si\ k = 1 < n \\ flip(T^R(n,k+1))\ o\ insert(T(n-1,k-1)) & si\ 1 < k < n \\ 1^n0^n & si\ k = n. \end{cases} \tag{7.1}$$

Le premier mot de $T(n,k)$ est :

$$first(T(n,k)) = \begin{cases} 101100(10)^{n-3} & si\ k = 1\ et\ n \geq 3 \\ 1^k010^k(10)^{n-k-1} & si\ 1 < k < n\ ou\ (k = 1\ et\ n = 2) \\ 1^n0^n & si\ k = n. \end{cases} \tag{7.2}$$

Le dernier mot de $T(n,k)$ est $1^k0^k(10)^{n-k}$.

Le premier mot de $T(n,k)$ nous permet de savoir d'où nous commençons la génération, et le dernier mot sert à arrêter cette génération. De plus, certaines listes commencent à partir de la fin d'autres. Par exemple, $T(5,2)$ commence par le mot flip(dernier mot de $T(5,3)$).

7.4 Algorithme non-récursif de Walsh

Notation 7.1 *Nous notons* 1_i*, la i-ième occurrence de 1 à partir de la gauche.*

Définition 7.3 *Par définition,* $1_1, 1_2, \cdots, 1_k$ *sont fixes et nous appelons les autres occurrences de* 1 ***libres****. La position la plus à droite que* 1_i *peut prendre est* $2i-1$*, et nous appelons un* 1 *libre qui n'est pas à sa position la plus à droite,* ***libéral****.*

Afin d'obtenir un algorithme non-récursif pour générer $T(n,k)$, Timothy Walsh (Walsh, 98) a généré les listes $T(n,k)$ pour $1 \leq k \leq 5$ à la main à partir de la définition récursive de l'équation 7.1, et il a obtenu les listes à la Fig. 7.1. Il a observé le patron suivi du mouvement de 1_i dans un intervalle de mots dans lequel $1_1, 1_2, \cdots, 1_{i-1}$ restent fixes, et a donné le théorème suivant.

Théorème 7.1 *L'ensemble de mots de* $T(n,k)$ *dans lesquels* $1_{k+1}, 1_{k+2}, \cdots, 1_{i-1}$ *restent fixes est un intervalle de mots consécutifs et il est partitionné en sous-intervalles dans lesquels* 1_i *est aussi fixe, et en passant d'un sous-intervalle à l'autre,* 1_i *se déplace une position à la fois vers la droite, de sa position la plus à gauche, adjacente à* 1_{i-1} *(sauf que* 1_{k+1} *commence à la position* $k+2$ *avec un zéro entre lui et* 1_k*), à sa position la plus à droite (position* $2i-1$ *dans le mot), ou vers la gauche, de sa position la plus à droite à celle la plus à gauche. Finalement,* 1_i *se déplace vers la droite si et seulement si le nombre de libéraux à sa gauche est pair.*

Exemple 7.1 *Dans* $T(5,2)$ *(Fig. 7.1), les 9 premiers mots avec le même préfixe* $1_1, 1_2, 1_3$*, forment un intervalle qui est divisé en 3 sous-intervalles : les 2 premiers mots, les 3 suivants et les 4 qui restent. Dans chaque intervalle,* 1_4 *est fixe et se déplace de sa position la plus à droite* $2 \times 4 - 1 = 7$ *à celle la plus à gauche, adjacente à* 1_3*, qui est* 5*, car le nombre de libéraux à gauche de* 1_4 *est* 1*, qui est impair.*

$T(1,1)$				
10				
$T(2,2)$	$T(2,1)$			
1100	1010			
$T(3,3)$	$T(3,2)$	$T(3,1)$		
111000	110100	101100		
	110010	101010		
$T(4,4)$	$T(4,3)$	$T(4,2)$	$T(4,1)$	
11110000	11101000	11010010	10110010	
	11100100	11010100	10110100	
	11100010	11011000	10111000	
		11001100	10101100	
		11001010	10101010	
$T(5,5)$	$T(5,4)$	$T(5,3)$	$T(5,2)$	$T(5,1)$
1111100000	1111010000	1110100010	1101001010	1011001010
	1111001000	1110100100	1101001100	1011001100
	1111000100	1110101000	1101011000	1011011000
	1111000010	1110110000	1101010100	1011010100
		1110010010	1101010010	1011010010
		1110010100	1101110000	1011110000
		1110011000	1101101000	1011101000
		1110001100	1101100100	1011100100
		1110001010	1101100010	1011100010
			1100110010	1010110010
			1100110100	1010110100
			1100111000	1010111000
			1100101100	1010101100
			1100101010	1010101010

Fig. 7.1 $T(n,k)$ pour $1 \leq k \leq 5$ à partir de l'équation 7.1.

Chercher le pivot — le plus grand i tel que g_i n'est pas à sa dernière valeur dans

$$s(g_{k+1} \cdots g_{i-1})$$

S'il existe un pivot i **alors**

changer g_i à sa valeur suivante dans $s(g_{k+1} \cdots g_{i-1})$;

changer chaque g_j, $i+1 \leq j \leq n$, à sa première valeur dans $s(g_{k+1} \cdots g_{j-1})$;

sinon $g_{k+1} \cdots g_n$ est le dernier mot dans la liste.

Fig. 7.2 Algorithme général pour trouver le mot suivant.

Définition 7.4 *Nous disons qu'une liste de mots est en* ***ordre graylex*** *si les mots qui possèdent le même suffixe (ou préfixe) sont générés de telle sorte qu'en passant d'un intervalle de mots possédant ce suffixe (ou préfixe) au suivant, les valeurs assumées par la lettre suivante soit augmentent de façon monotone, soit diminuent de façon monotone (Chase, 89).*

Corollaire 7.1 *Soit g_i, la position de 1_i dans un mot de $T(n,k)$. Alors, la liste des mots correspondant $g_{k+1} \cdots g_n$ a la propriété que l'ensemble des mots qui ont le même préfixe $g_{k+1} \cdots g_{i-1}$ est un intervalle de mots consécutifs et est partitionné en sous-intervalles dans lesquels g_i est fixe et en passant d'un sous-intervalle à l'autre, la suite $s(g_{k+1} \cdots g_{i-1})$ de valeurs assumées par g_i est donnée par :*

$$s() = (k+2, k+3, \cdots, 2k+1) \qquad \textit{préfixe vide, donc valeurs de } g_{k+1}$$

$$s(g_{k+1}g_{k+2} \cdots g_{i-1}) = \begin{cases} (g_{i-1}+1, \cdots, 2i-1) & \textit{si } g_j < 2j-1 \\ & \textit{pour un nombre pair de } j,\ k+1 \leq j \leq i-1, \\ (2i-1, \cdots, g_{i-1}+1) & \textit{si } g_j < 2j-1 \\ & \textit{pour un nombre impair de } j,\ k+1 \leq j \leq i-1. \end{cases}$$

Comme $s(p)$ est monotone pour n'importe quel préfixe p, la liste des mots $g_{k+1} \cdots g_n$ est en ordre graylex (voir la définition 7.4). Le successeur d'un mot donné est déterminé par l'algorithme général de la Fig. 7.2.

$$\begin{array}{llll} \text{AVANT} & \overrightarrow{1_i}\overleftarrow{1}00\cdots000\overrightarrow{1}0\overrightarrow{1}0\cdots\overrightarrow{1}0 & \text{AVANT} & \overrightarrow{1_i}\overleftarrow{1}000\overrightarrow{1}0\overrightarrow{1}0\cdots\overrightarrow{1}0 \\ \text{APRÈS} & \overrightarrow{1_i}00\cdots0\overleftarrow{1}0\overleftarrow{1}0\overleftarrow{1}0\cdots\overleftarrow{1}0 & \text{APRÈS} & \overrightarrow{1_i}\overrightarrow{1}00\overleftarrow{1}0\overleftarrow{1}0\cdots\overleftarrow{1}0 \end{array} \quad (1_i \textit{ à sa plus droite})$$

$$\begin{array}{llll} \text{AVANT} & \overleftarrow{1_i}00\cdots0\overrightarrow{1}0\overrightarrow{1}0\overrightarrow{1}0\cdots\overrightarrow{1}0 & \text{AVANT} & \overleftarrow{1_i}\overleftarrow{1}00\overrightarrow{1}0\overrightarrow{1}0\cdots\overrightarrow{1}0 \\ \text{APRÈS} & \overleftarrow{1_i}\overrightarrow{1}00\cdots000\overleftarrow{1}0\overleftarrow{1}0\cdots\overleftarrow{1}0 & \text{APRÈS} & \overleftarrow{1_i}\overrightarrow{1}000\overleftarrow{1}0\overleftarrow{1}0\cdots\overleftarrow{1}0 \end{array} \quad (1_i \textit{ à sa plus droite})$$

Fig. 7.3 Suffixe commençant par 1_i avant et après le mouvement de 1_i. Nous avons 4 cas. La flèche indique la direction du mouvement.

Au lieu de stocker un tableau auxiliaire $g_{k+1}\cdots g_n$, T. Walsh (Walsh, 98) a modifié l'algorithme général de telle sorte qu'il agit directement sur chaque mot de Dyck. Pour tout i de n à $k+1$, s'il y a un nombre pair de libéraux à gauche de 1_i, alors 1_i se déplace vers la droite et sa dernière position est $2i-1$. Sinon 1_i se déplace vers la gauche et sa dernière position est adjacente à 1_{i-1} (1_{k+1} ne se déplace jamais vers la gauche).
Si tous les 1 sont à leurs dernières positions, alors le mot est le dernier. Sinon, le pivot est l'indice i du premier 1_i trouvé qui n'est pas à sa dernière position ; 1_i se déplace une position à droite ou à gauche selon la direction dans laquelle il se déplaçait, et si $i < n$, alors tous les 1 à sa droite doivent être déplacés à leurs premières positions.
En examinant la Fig. 7.3, nous voyons que seuls 1_i et 1_{i+1} doivent se déplacer ; tout 1_j, $j > i+1$, est à sa position la plus à droite, qui était sa dernière position et maintenant, est devenue sa première après que 1_i et 1_{i+1} se soient déplacés, car le nombre de libéraux à sa gauche change toujours par 1.

7.4.1 Implantation de l'algorithme de Walsh

Afin que l'algorithme s'exécute en $O(n)$, T. Walsh (Walsh, 98) maintient une variable booléenne *Odd* qui est vraie si le nombre total de libéraux dans un mot donné est impair. Avant de fouiller un mot, il initialise une variable booléenne $Left$ à *Odd* et il l'inverse chaque fois qu'un libéral est rencontré, de telle sorte que chaque 1 est examiné, la variable $Left$ est vraie si ce 1 se déplace vers la gauche. Il maintient aussi une variable

booléenne *Last* qui devient vraie si le mot courant est le dernier dans $T(n,k)$.
L'algorithme qui donne le mot suivant se trouve à la Fig. 7.4, ce pseudo-code vient directement de (Walsh, 98) ; seuls les commentaires ont été traduits en français.

Si nous voulons générer $T(n,k)$, alors nous générons $first(T(n,k))$ de l'équation 7.2, nous initialisons *Last* à $faux$ et nous initialisons *Odd* à *vraie* si $n > 2$ et $k < n$, car selon l'équation 7.2, $first(T(n,k))$ possède un seul libéral $(1_{max(3,k+1)})$. Sinon, nous mettons *Odd* à $faux$, car $first(T(n,k))$ ne possède pas de libéral. Nous exécutons alors l'algorithme *Next* de la Fig. 7.4 jusqu'à ce que *Last* soit *vraie*.

Si nous voulons générer tous les mots de Dyck de longueur $2n$, il suffit de générer $T(n+1,1)$, puis nous éliminons le préfixe 10, ou de façon équivalente, générer $T(n,n)oT(n,n-1)o\cdots oT(n,2)oT^R(n,1)$. Pour cela, il suffit de commencer par générer $T(n,n)$ avec les changements (Fig. 7.5) apportés à l'algorithme Next de la Fig. 7.4.

7.5 Exemples

Exemple 7.2 *Nous allons montrer quelques étapes de l'exécution de l'algorithme Next de la Fig. 7.4, pour la génération de la liste* $T(5,2)$.
Il faut d'abord trouver $first(T(5,2))$ *selon l'équation 7.2 (*$n = 5$ *et* $k = 2$*) :*
1 1 0 1 0 0 1 0 1 0.
L'initialisation : Last :=false et Odd :=true. La trace est dans le Tab. 7.1.

Exemple 7.3 *Pour générer tous les mots de Dyck de longueur 8, il suffit de trouver* $T(4,4)oT(4,3)oT(4,2)oT^R(4,1)$. *Pour cela, nous commençons avec le mot : 1 1 1 1 0 0 0 0 et nous appliquons Next avec les changements de la Fig. 7.5. Nous obtenons la trace qui est dans le Tab. 7.2.*

En comparant le Tab. 7.2 avec $T(4,4), T(4,3), T(4,2)$ et $T^R(4,1)$ de la Fig. 7.1, nous voyons la similarité.

```
1. Procedure Next(n, k :integer ; p :array[1..2n] ; Odd, Last : boolean)
2.   i := n ; {Le 1 (suivant) rencontré pendant une fouille de droite à gauche sera 1_i.}
3.   j := 2n − 1 ; {j est la position de 1_i dans le mot. }
4.   Left :=Odd ; {Odd est vrai s'il y a un nombre impair de libéraux dans le mot.}
5.   while i > k do
6.     if p[j] = 1 then
7.       if j < 2i − 1 then {1_i est un libéral.} Left :=not Left end if ;
8.       if Left then {Il y a un nombre impair de libéraux à gauche de 1_i.}
9.         if p[j − 1] = 0 and j > k + 2 then {1_i peut bouger vers la gauche.}
10.            p[j − 1] := 1 ;
11.            if i = n then
12.              p[j] := 0 ; {1_i échange avec le 0 à sa gauche.}
13.              if j = 2i − 1 then Odd :=not Odd end if {1_i est devenu un libéral.}
14.            else {Voir la Fig. 7.3}
15.              if j < 2i − 1 then p[2i + 1] := 0 else p[j + 1] := 0 end if ;
16.              Odd :=not Odd {Soit 1_i, soit 1_{i+1} est devenu un libéral.}
17.            end if ;
18.            return
19.        end if {Sinon 1_i est à sa dernière position et la fouille continue.}
20.      else {Il y a un nombre pair de libéraux à gauche de 1_i.}
21.        if j < 2i − 1 then {1_i peut bouger vers la droite.}
22.          p[j] := 0 ;
23.          if i = n then
24.            p[j + 1] := 1 ; {1_i échange avec le 0 à sa droite.}
25.            if j = 2i − 2 then Odd :=not Odd end if {1_i n'est plus un libéral.}
26.          else {Voir la Fig. 7.3}
27.            if j < 2i − 2 then p[2i + 1] := 1 else p[j + 2] := 1 end if ;
28.            Odd :=not Odd {Soit 1_i, soit 1_{i+1} n'est plus un libéral.}
29.          end if ;
30.          return
31.        end if {Sinon 1_i est à sa dernière position et la fouille continue.}
32.      end if ;
33.      i := i − 1
34.    end if ; {Sinon p[j] = 0 et est ignoré.}
35.    j := j − 1
36.  end while ; {Tous les libres sont à leurs dernières positions.}
37.  Last :=true ; { p est le dernier mot dans la liste T(n, k) − ou dans la liste T^R(n, 1).}
38. end Next.
```

Fig. 7.4 Algorithme pour trouver le successeur d'un mot donné de $T(n,k)$ en temps $O(n)$ et espace auxiliaire $O(1)$.

changer la ligne 5 à "**while** $i \geq 1$ **do**" ;

changer à la ligne 7, "**if** $j < 2i - 1$" à "**if** $j < 2i - 1$ **and** $i > k$" ;

insérer le code suivant après la ligne 22 :

if $i = k$ **then** {Nous passons de $T(n,k)$ à $T(n,k-1)$ ou à $T^R(n,1)$.}

$p[j+1] := 1; k := k - 1$; {Seulement 1_k bouge.}

Odd := **not** Odd ; {Les directions du mouvement sont renversées pour tout k ;

si $k > 1$, alors 1_{k+1} est un nouveau libéral ;

si $k = 1$, alors la liste est renversée.}

return

end if ;

Fig. 7.5 Changements apportés à l'algorithme Next (Fig. 7.4)

itération	i	j	Left	Odd	Last	1	2	3	4	5	6	7	8	8	10
0				true	false	1	1	0	1	0	0	1	0	1	0
1	5	9	true	false									1	0	
2	5	9	false								1		0		
		8	true												
	4	7		true											
3	5	9	true									0	1		
		8													
		7	false												
4															
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Tab. 7.1 Quelques étapes de la trace de l'algorithme de la Fig. 7.4 pour la génération de $T(5,2)$; dans ce tableau, à l'exception de l'itération 0, une case est vide lorsqu'elle n'a pas changé de valeur.

$$
\begin{array}{lcl}
T(4,4) & \rightarrow & 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\
T(4,3) & \rightarrow & \left\{ \begin{array}{cccccccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array} \right. \\
T(4,2) & \rightarrow & \left\{ \begin{array}{cccccccc} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{array} \right. \\
T^R(4,1) & \rightarrow & \left\{ \begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array} \right.
\end{array}
$$

Tab. 7.2 Trace de l'algorithme Next (Fig. 7.4) avec les changements de la Fig. 7.5 pour trouver $T(4,4)oT(4,3)oT(4,2)oT^R(4,1)$.

CHAPITRE VIII

CODE GRAY POUR LES INVOLUTIONS SANS POINT FIXE

8.1 Introduction

Les algorithmes pour générer toutes les involutions de longueur n en ordre lexicographique sont présentés dans (Foundas, 91; Roelants van Baronaigien, 92) ; le premier article contient aussi des algorithmes pour générer toutes les involutions sans point fixe de longueur $2n$ en ordre lexicographique.

Timothy Walsh (Walsh, 2001) a utilisé les I-codes (voir le paragraphe après la définition 8.2) de Roelants van Baronaigien (Roelants van Baronaigien, 92) pour les involutions pour dériver un code Gray pour les involutions sans point fixe de longueurs $2n$ de telle sorte que chaque involution est transformée en son successeur par deux transpositions. Il a présenté un algorithme qui permet de passer d'un mot à sa position dans la liste et vice versa, ainsi qu'un algorithme non-récursif qui permet de trouver le successeur d'un mot ou de déterminer si le mot est le dernier dans sa liste. Ce dernier algorithme fonctionne en temps $O(n)$ au pire des cas et utilise $O(1)$ variables auxiliaires.

T. Walsh a généralisé la méthode de Chase pour obtenir ce dernier algorithme pour n'importe quelle liste de mots pour laquelle tous les mots qui possèdent le même suffixe forment un intervalle de mots consécutifs. De plus, il a montré que si la lettre qui précède ce suffixe prend toujours au moins 2 valeurs distinctes dans cet intervalle, alors la méthode d'Ehrlich trouve en temps $O(1)$ la lettre la plus à droite pour laquelle un mot diffère de son successeur.

8.2 Terminologie et définitions

Définition 8.1 *Une **involution** de longueur n est une permutation $(p_1, p_2, \cdots, p_n)$ de $\{1, \cdots, n\}$ telle que pour tout i, $1 \leq i \leq n$, $p_i = i$, ou $p_i = j \neq i$ et $p_j = i$. Un élément p_i est un **point fixe** de l'involution si $p_i = i$.*

Exemple 8.1 *La permutation $(4, 2, 3, 1, 7, 6, 5, 8) = (1, 4)(2)(3)(5, 7)(6)(8)$ est une involution ayant comme points fixes : $2, 3, 6$ et 8.*

Une **FPFI** (fixed-point-free involution) est une involution sans point fixe.

Définition 8.2 *Le **rang** d'un élément w dans une liste L est le nombre d'éléments qui précèdent w dans L.*

Bijection préservant l'ordre lexicographique. Les I-codes pour les involutions (Roelants van Baronaigien, 92) restreints aux FPFI est une bijection qui préserve l'ordre lexicographique de l'ensemble de $(2n-1) \times (2n-3) \times \cdots \times 3 \times 1$ de FPFI de longueur $2n$ en notation en une ligne dans le produit cartésien des intervalles des entiers $[1, 2n-1] \times [1, 2n-3] \times \cdots \times [1, 3] \times [1, 1]$. En soustrayant 1 de chaque n-uplet, puis en éliminant son dernier membre qui est 0, nous obtenons la bijection R qui préserve l'ordre lexicographique de l'ensemble de FPFI de longueur $2n$ dans $[0, 2n-2] \times [0, 2n-4] \times \cdots \times [0, 2]$.

Définition 8.3 *La **notation canonique** en cycles d'une FPFI est $(x(1), y(1)) \cdots (x(n), y(n))$, où $x(i) < y(i)$ pour tout i, et $x(1) < \cdots < x(n)$.*

Remarque 8.1 *Pour tout i, $x(i)$ est le plus petit élément de $\{1, \cdots, 2n\} - \{x(1), y(1), \cdots x(i-1), y(i-1)\}$ et il y a $2n - 2i + 1$ éléments parmi lesquels nous pouvons choisir $y(i)$ de l'ensemble :*

$$S(i) = \{1, \cdots, 2n\} - \{x(1), y(1), \cdots, x(i-1), y(i-1), x(i)\}.$$

ordre lexicographique			*ordre code Gray*		
cycles	F	$R(F)$	*cycles*	F	$R(F)$
(12)(34)(56)	214365	00	(12)(34)(56)	214365	00
(12)(35)(46)	215634	01	(13)(24)(56)	341265	10
(12)(36)(45)	216543	02	(14)(23)(56)	432165	20
(13)(24)(56)	341265	10	(15)(23)(46)	532614	30
(13)(25)(46)	351624	11	(16)(23)(45)	632541	40
(13)(26)(45)	361542	12	(16)(24)(35)	645231	41
(14)(23)(56)	432165	20	(15)(24)(36)	546213	31
(14)(25)(36)	456123	21	(14)(25)(36)	456123	21
(14)(26)(35)	465132	22	(13)(25)(46)	351624	11
(15)(23)(46)	532614	30	(12)(35)(46)	215634	01
(15)(24)(36)	546213	31	(12)(36)(45)	216543	02
(15)(26)(34)	564312	32	(13)(26)(45)	361542	12
(16)(23)(45)	632541	40	(14)(26)(35)	465132	22
(16)(24)(35)	645231	41	(15)(26)(34)	564312	32
(16)(25)(34)	654321	42	(16)(25)(34)	654321	42

Tab. 8.1 Les FPFI F de $\{1,\cdots,6\}$ et leurs I-codes $R(F)$ en ordre lexicographique et code Gray.

La fonction R fait correspondre à la FPFI $(x(1),y(1))\cdots(x(n),y(n))$ la suite $(g_1,\cdots,g_{n-1})$, où g_i est le nombre d'éléments de $S(i)$ qui sont $< y(i)$ (voir la partie gauche du Tab. 8.1). Le rang lexicographique du mot $(g_1,\cdots,g_{n-1})$ est exactement le nombre $g_1\cdots g_{n-1}$, où chaque g_i est en base $2n-2i+1$. Passer d'un I-code à son rang se fait en $O(n)$ opérations arithmétiques. Passer d'une FPFI F à $R(F)$ prend $O(n^2)$ opérations élémentaires si la suppression d'un élément de $S(i)$ prend $O(n)$. Ce coût peut être réduit à $O(n\log n)$ si l'ensemble $S(i)$ est mis à jour en utilisant un arbre équilibré ou les algorithmes les plus efficaces en espace ((Knuth, 73), les réponses aux exercises 5 et 6, à la page 9) et à

$O(\frac{n \log n}{\log \log n})$ en utilisant une structure de données sophistiquée (Dietz, 89).

Exemple 8.2 *Soit* $F = (f_1, \cdots, f_n) = (7, 4, 6, 2, 8, 3, 1, 5)$.
Nous voyons que $x(1) = 1$ *et* $S(1) = \{2, 3, 4, 5, 6, 7, 8\}$. *Comme* $y(1) = f_1 = 7$, *qui est supérieur à 5 éléments de* $S(1)$, *alors* $g_1 = 5$. *Ainsi,* $x(2) = 2$ *et* $S(2) = \{3, 4, 5, 6, 8\}$. *Comme* $y(2) = f_2 = 4$, *alors* $g_2 = 1$. *Puis,* $x(3) = 3$ *et* $S(3) = \{5, 6, 8\}$. *Comme* $y(3) = f_3 = 6$, *alors* $g_3 = 1$. *D'où* $R(F) = (5, 1, 1)$, *alors le rang lexicographique de* F *est 511 en base* $(7, 5, 3)$ *qui est* $((5 \times 3) + 1) \times 5 + 1 = 81$.
Réciproquement, connaissant le rang 81 de F, *nous pouvons extraire* $R(F) = (5, 1, 1)$ *et par des calculs successifs de* $S(i)$, *nous pouvons trouver* F.

8.3 Algorithmes de Walsh

T. Walsh a présenté un code Gray (Walsh, 2001) pour FPFI tel que chaque FPFI est transformée à la suivante par deux transpositions – deux cycles (x, y) et (i, j) avec $x < y$ et $|y - j| = 1$ sont remplacés par les cycles (x, j) et (i, y) – alors en notation en une ligne, $p_y = x$ échange avec son voisin immédiat $p_j = i$, et $p_x = y$ échange avec $p_i = j$.

Remarque 8.2 *La fonction* R *préserve aussi bien le voisinage que l'ordre lexicographique : si* g_i *est augmenté de 1 alors* $y(i)$ *doit augmenter à la valeur plus grande suivante* j *dans* $S(i)$. *En notation cyclique,* $(x(i), y(i))$ *et* (j, p_j) *sont remplacés par* $(x(i), j)$ *et* $(y(i), p_j)$. *En notation en une ligne,* $y(i)$ *échange avec* j, *alors que* $x(i)$ *échange avec* p_j, *l'élément le plus proche à droite de* $x(i)$, *qui est dans* $S(i)$, *ou de façon équivalente qui est supérieur à* $x(i)$.
Walsh a choisi le code Gray pour lequel g_1 *varie le plus vite ; alors quand* g_i *change, les* g_i, $j < i$, *sont tous à leurs valeurs extrêmes, de telle sorte que* $S(i)$ *est un intervalle. Ainsi,* $y(i)$ *doit changer par 1 et* $x(i)$ *doit échanger avec son voisin immédiat. D'où il a défini* $\mathcal{F}(n)$ *comme le code Gray* $(g_1, \cdots, g_{n-1})$, *où* $0 \leq g_i \leq 2(n-1)$ *induit sur l'ensemble de FPFI de* $\{1, \cdots, 2n\}$ *(voir la partie droite du Tab. 8.1).*
La première valeur de F *est* $(2, 1, 4, 3, \cdots, 2n, 2n-1)$ *et* $R(F) = (0, \cdots, 0)$.

8.3.1 Algorithme 1 pour trouver le prochain mot d'une liste

T. Walsh a remarqué (Walsh, 2001) que tous les mots qui possèdent un même suffixe $(g_{i+1}, \cdots, g_{n-1})$ forment un intervalle de mots consécutifs dans lesquels g_i prend la suite des valeurs distinctes $(0, 1, \cdots, 2(n-i))$ si $g_{i+1} + \cdots + g_{n-1}$ est pair, et $(2(n-i), \cdots, 1, 0)$ sinon. Dans le premier cas, nous disons que g_i est croissant et dans le deuxième cas, décroissant, ce qui lui a permis de généraliser l'algorithme de séquence utilisé dans (Chase, 89; Williamson, 85) pour obtenir l'algorithme 1 de la Fig. 8.1.
Cet algorithme prend comme premier mot $(g_1, \cdots, g_{n-1}) = (0, \cdots, 0)$ et fonctionne pour n'importe quelle liste de mots $(g_1, \cdots, g_{n-1})$ dans laquelle tous les mots qui possèdent un même suffixe $(g_{i+1}, \cdots, g_{n-1})$ forment un intervalle de mots consécutifs dans la liste, de telle sorte que cet intervalle est partitionné en sous-intervalles par des suites de valeurs distinctes assumées par g_i.

Chercher g_i le plus à gauche, qui n'est pas à sa dernière valeur dans sa suite ;
si un tel g_i est trouvé **alors** {i est le pivot}
    Changer g_i à la valeur suivante dans sa suite ;
    Changer chaque $g_j, j < i$, qui n'est pas à sa première valeur dans sa suite
    à la première valeur ;
sinon le mot courant est le dernier.

Fig. 8.1 Algorithme 1. L'algorithme général de séquence pour transformer le mot $(g_1, \cdots, g_{n-1})$ en son successeur.

Remarque 8.3 *D'après la définition 6.7 et comme la liste de mots pour ce code est en partition suffixe, alors pour chaque mot w dans cette liste, sauf pour le dernier, le* ***pivot*** *est l'indice de la lettre la plus à droite (l'élément pivot) qui doit changer pour transformer w en son successeur.*

Dans le code Gray du Tab. 8.1, seulement g_i change – tous les $g_j, j < i$, sont maintenant à leur permière valeur par rapport au nouveau suffixe – et ce changement pour $R(F) = (g_1, \cdots, g_{n-1})$ induit une paire de transpositions correspondantes dans la FPFI F.

Exemple 8.3 *Si* $F = (8,4,6,2,7,3,5,1)$ *ou de façon équivalente* $(18)(24)(36)(57)$, *alors* $R(F) = (g_1, g_2, g_3) = (6,1,1)$. *Comme* $g_2 + g_3$ *est paire,* g_1 *prend la suite des valeurs* $(0,1,\cdots,6)$ *; alors* g_1 *est déjà à sa dernière valeur. Comme* g_3 *est impair,* g_2 *prend la suite des valeurs* $(4,3,2,1,0)$ *; alors 2 est le pivot et* g_2 *diminue à 0, changeant* $R(F)$ *à* $(6,0,1)$ *et* F *à* $(8,3,2,6,7,4,5,1)$ *ou de façon équivalente* $(18)(23)(46)(57)$.

Exemple 8.4 *Si* $F = (2,1,4,3,7,8,5,6)$ *alors* $R(F) = (g_1, g_2, g_3) = (0,0,1)$. *Comme* $g_2 + g_3$ *est impair,* g_1 *prend la suite des valeurs* $(6,5,\cdots,0)$ *; alors* g_1 *est déjà à sa dernière valeur. Comme* g_3 *est impair,* g_2 *prend la suite des valeurs* $(4,3,2,1,0)$ *; alors* g_2 *est aussi à sa dernière valeur. Le suffixe pour* g_3 *est vide, ayant comme somme 0, alors* g_3 *prend la suite des valeurs* $(0,1,2)$. *D'où 3 est le pivot et* g_3 *augmente à 2, changeant* $R(F)$ *à* $(0,0,2)$ *et* F *à* $(2,1,4,3,8,7,6,5)$.

8.3.2 Algorithme 2 pour trouver le prochain mot d'une liste en temps $O(1)$ dans le pire cas

T. Walsh a optimisé l'algorithme de séquence (Fig. 8.1) pour éviter de calculer la parité de $g_{i+1} + \cdots + g_{n-1}$ pour chaque nouveau i. Il a observé que pour chaque $j < i, g_j$ est 0 ou $2(n-j)$, qui est pair dans les deux cas ; alors g_i est croissant si et seulement s'il a la même parité que $g_i + \cdots + g_{n-1}$. Alors il stocke une variable booléenne *Odd*, qui est vraie si $g_i + \cdots + g_{n-1}$ est impair ; *Odd* est initialisée à fausse pour le premier mot $(0,\cdots,0)$ et change de valeur pour chaque mot successif.

Maintenant, la première valeur, la suivante et la dernière de chaque g_i peut être calculée en temps $O(1)$, de telle sorte que l'algorithme de séquence (Fig. 8.1), pour une FPFI F de longueur n, fonctionne en temps $O(n)$ et utilise un tableau auxiliaire de taille n pour $R(F)$.

Le théorème 8.1 suivant (Walsh, 2001) nous évite de stocker $R(F)$ comme un tableau auxiliaire afin d'obtenir un algorithme de séquence pour $\mathcal{F}(2n)$.

Théorème 8.1 *Soient* $F = (f_1, \cdots, f_n)$ *et* i *le pivot de* $R(F)$. *Si Odd est vraie, alors* $x(i) = 2i - 1, g_i = y(i) - 2i, F = (2, 1, 4, 3, \cdots, 2i - 2, 2i - 3, y(i) > 2i, \cdots)$ *et* i *est la plus petite valeur telle que* $f_{2i-1} > 2i$ *; sinon* $x(i) = i, g_i = y(i) - (i + 1), F = (2n, 2n - 1, \cdots, 2n - i + 2, y(i) < 2n - i + 1, \cdots, i - 1, \cdots, 2, 1)$ *et* i *est la plus petite valeur telle que* $f_i < 2n - (i - 1)$ *s'il existe.*

Exemple 8.5 *Dans l'exemple 8.3,* $F = (8, 4, 6, 2, 7, 3, 5, 1)$ *et* $R(F) = (6, 1, 1)$ *(qui n'est pas stocké), alors Odd est fausse. Maintenant,* $f_1 = 8$, *mais* $f_2 < 7$ *; alors* $i = 2$. *D'où* $x(2) = 2, y(2) = f_2 = 4$ *et* $g_2 = 4 - (2 + 1) = 1$ *qui est impair; alors il est décroissant. Par conséquent,* $f_4 = 2$ *échange avec son voisin immédiat de gauche* $f_3 = 6$ *et leurs associés* $f_2 = 4$ *et* $f_6 = 3$ *aussi s'échangent, d'où* F *change à* $(8, 3, 2, 6, 7, 4, 5, 1)$.

Exemple 8.6 *Dans l'exemple 8.4,* $F = (2, 1, 4, 3, 7, 8, 5, 6)$ *et* $R(F) = (0, 0, 1)$ *(qui n'est pas stocké), alors Odd est vraie. Maintenant,* $f_1 = 2$ *et* $f_3 = 4$, *mais* $f_5 > 6$ *; alors* $i = 3$. *D'où* $x(3) = 5, y(3) = f_5 = 7$ *et* $g_3 = 7 - (2 \times 3) = 1$ *qui est impair; alors il est croissant. Par conséquent,* $f_7 = 5$ *échange avec son voisin immédiat de droite* $f_8 = 6$ *et leurs associés* $f_5 = 7$ *et* $f_6 = 8$ *aussi s'échangent, d'où* F *change à* $(2, 1, 4, 3, 8, 7, 6, 5)$.

L'algorithme maintient seulement F, *Odd* et une autre variable booléenne *Done* qui est initialisée à fausse et devient vraie lorsque la FPFI devient la dernière; il utilise donc $O(1)$ variables auxiliaires, mais prend un temps $O(n)$ dans le pire des cas pour trouver le pivot i en fouillant F de gauche à droite. Cette fouille peut être évitée en utilisant le tableau auxiliaire $e = (e_1, \cdots, e_n)$ introduit dans (Bitner, Ehrlich et Reingold, 76; Ehrlich, 73) et généralisé dans ((Williamson, 85), page 112) et ailleurs.

T. Walsh a généralisé encore la méthode de (Bitner, Ehrlich et Reingold, 76) pour qu'elle fonctionne sur n'importe quelle liste de mots dans laquelle tous les mots qui possèdent un suffixe en commun, forment un intervalle de mots consécutifs pour lesquels la lettre

{*Avant l'exécution :*}
{*pour tout j, e[j]=j, sauf si (g[j], ··· , g[k-1]) est un z-mot pour un certain k>j,*}
{*et dans ce cas e[j]=k.*}
$i := e[1]$;
if $i = n$ **then** {*(g[1], ···, g[n-1])=(z[1], ··· , z[n-1])*}
Done :=**True** ; **return**
end if ;
{*Sinon (g[1], ···, g[i-1])=(z[1], ··· , z[i-1]), mais g[i]≠ z[i], alors i est le pivot.*}
{*Dans l'un ou l'autre cas, (e[1], ··· ,e[i-1],e[i])=(i,2,3, ··· ,i-1,i).*}
$(g[1], \cdots, g[i-1], g[i]) = (a[1], \cdots, a[i-1], h[i])$;
{*O(1) changements dans le cas d'un code Gray*}
update any other auxiliary variables ;
$e[1] := 1$;
if $g[i] = z[i]$ **then** {*Ou bien e[i+1]=i+1, g[i+1]≠ z[i+1] et (z[i]) est un z-mot*}
{*ou e[i+1]=k+1>i+1, (g[i+1], ···, g[k]) était un z-mot*}
{*et maintenant (g[i],g[i+1], ···, g[k]) est un z-mot.*}
$e[i] := e[i+1]$;
{*Maintenant (g[i], ···, g[k]) est un z-mot et e[i]=k+1 si k=i ou sinon.*}
$e[i] := i+1$ {*Comme $g[i] = z[i], g[i+1]$ ne peut pas commencer un z-mot.*}
end if.
{*Sinon, ou bien i=1 ou g[1]=a[1]*}
{*et dans l'un ou l'autre cas g[1]≠ z[1] et e[1] est toujours 1.*}
{*Après l'exécution :*}
{*pour tout j, e[j]=j sauf si (g[j], ··· , g[k-1]) est un z-mot pour un certain z>j,*}
{*et dans ce cas e[j]=k.*}

Fig. 8.2 Algorithme 2. L'algorithme général sans boucle pour trouver les suites pour $(g[1], \cdots, g[n-1])$, en utilisant le tableau $(e[1], \cdots, e[n])$. $a[i], z[i]$ et $h[i]$ sont respectivement, la première valeur, la dernière et celle suivant $g[i]$. Initialement $g[i] = a[j]$ et $e[j] = j$ pour tout j.

qui précède le suffixe prend au moins deux valeurs distinctes (voir la Fig 8.2).
Notons a_i, z_i et h_i respectivement, la première , la dernière et la valeur suivante de g_i dans la suite de valeurs distinctes assumées par g_i comme fonction du suffixe $(g_{i+1}, \cdots, g_{n-1})$. Comme g_i prend au moins deux valeurs distinctes, a_i n'est jamais égale à z_i.

Définition 8.4 *Un* ***z-mot*** *est un sous-mot* $(g_j, \cdots, g_k) = (z_j, \cdots, z_k)$ *qui est maximal par inclusion : ou bien* $j = 1$ *ou* $g_{j-1} \neq z_{j-1}$, *et, ou bien* $k = n - 1$ *ou* $g_{k+1} \neq z_{k+1}$.

De l'algorithme 1 (Fig. 8.1), il est clair que le pivot est la valeur la plus petite de i telle que $g_j \neq z_i$. Si $g_1 \neq z_1$, alors 1 est le pivot. Si $(g_1, \cdots, g_{i-1})$ est un z-mot où $i < n$, alors i est le pivot. Si $(g_1, \cdots, g_{n-1}) = (z_1, \cdots, z_{n-1})$, alors c'est le dernier mot dans la liste et nous appelons n le pivot. Nous allons voir que pour tout j, $e_j = j$, sauf si $(g_j, \cdots, g_{k-1})$ est un z-mot pour un certain $k > j$; dans ce cas $e_j = k$. Ainsi, e_1 est toujours le pivot. Le premier mot est $(a_1, \cdots, a_{n-1})$ et comme $a_i \neq z_i$ pour tout i, il n'y a aucun z-mot. $(e_1, \cdots, e_n)$ est initialisé à $(1, \cdots, n)$.
La liste de I-codes pour FPFI satisfait la condition sous laquelle l'algorithme 2 (Fig. 8.2) fonctionne. En utilisant le théorème 8.1, T. Walsh a implanté un algorithme de séquence sans boucle pour FPFI, sans stocker le tableau auxiliaire $R(F)$; c'est l'algorithme 3 à la Fig. 8.3.

8.4 Exemple

La trace (quelques étapes) de l'exécution de l'algorithme 3 (Fig. 8.3) pour $F = (2, 1, 4, 3, 6, 5)$, $n = 3$, est au Tab. 8.2.

$i := e[1]$; { i est le pivot dans $R(F) = (g[1], \cdots, g[n-1])$ }
if $i = n$ **then** { F est la dernière FPFI }
    Done :=**True** ; **return**
end if ; { dans ce qui suit, $x = x(i), y = y(i), g = g[i]$, }
{ et Up est vrai si $g[i]$ est croissant }
if Odd **then** { $R(F)$ a une somme impaire }
    $x := 2 * i - 1; y := f[x]; g := y - 2 * i; Up :=$ (g is odd)
else
    $x := i; y := f[x]; g := y - (i + 1); Up :=$ (g is even)
end if ;
if (Up) **then** { $g[i]$ doit augmenter }
    $g := g + 1; j := y + 1$; { $f[j]$ est le voisin de droite de $f[y] = x$ }
else { $g[i]$ doit diminuer }
    $g := g - 1; j := y - 1$; { $f[j]$ est le voisin de gauche de $f[y] = x$ }
end if ;
swap $f[y] = x$ with $f[j]$; swap $f[x] = y$ with $f[f[j]] = j$
Odd :=**not**(Odd) ;
e[1] :=1 ;
if $(g = 0)$ or $(g = 2 * (n - i))$ **then** { le nouveau $g[i]$ est à sa dernière valeur }
    $e[i] := e[i + 1]; e[i + 1] := i + 1$
end if.

Fig. 8.3 Algorithme 3. L'algorithme de séquence en temps $O(1)$ pour la FPFI $F = (f[1], \cdots, f[2n])$, en utilisant seulement le tableau e de taille n.

i	$x[i]$	$y[i]$	j	Odd	Up	e						g		F						$Done$
$init$				$false$		1	2	3	4	5	6	0	0	2	1	4	3	6	5	$false$
1	1	2	3	$true$	$true$	1						0, 1		3	4	1	2	6	5	
1	1	3	4	$false$	$true$	1						1, 2		4	3	2	1	6	5	
1	1	4	5	$true$	$true$	1						2, 3		5	3	2	6	1	4	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Tab. 8.2 Trace de l'algorithme 3 (Fig. 8.3) pour $F = (2, 1, 4, 3, 6, 5)$, $n = 3$.

CONCLUSION

Dans ce mémoire, nous avons présenté une implantation non-récursive du code Gray de Pruesse et Ruskey pour les idéaux d'un poset. Notre implantation a la même complexité asymptotique que l'implantation récursive de Pruesse et Ruskey, mais elle s'exécute un peu plus rapidement. Il serait intéressant de trouver une implantation sans boucle pour ce code Gray, ou pour un autre, pour les idéaux d'un poset, ou au moins une implantation dont la complexité est compétitive avec celle de la génération en ordre lexicographique de Squire. Ce problème sera l'un des sujets de notre recherche dans l'avenir.

Nous avons également donné un compte rendu de plusieurs codes Gray dans la littérature et leur implantation. En particulier, nous avons donné une preuve formelle de la validité du programme FORTRAN de Chase qui génère son code Gray pour les combinaisons.

APPENDICE A

PROGRAMME

```
// Le programme suivant implante l'algorithme non-récursif de la Fig. 1.12.
// Il est écrit en C++.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_POINTS 50   // La taille maximale de l'ensemble.
#define MAXSTACK 50
typedef enum{FORWARD, BACKWARD}Direction;
                // Pour connaître la direction de la fouille du graphe courant.
unsigned int table[100];
                // table[1] = 01010, signifie 1<3. Au début, le tableau contient seulement
                // la description du poset. Il contiendra la clôture transitive après l'appel
                // de la procédure Process().
unsigned int up[100];
                // up[i] contient les éléments qui sont plus grands que i dans le poset.
                // Par exemple, up[3] = 01101000 signifie que 3<5 et 3<6.
unsigned int steiner[100];
                //steiner[3] = 2, signifie que l'élément 3 a deux éléments plus petits que lui.
int flip;       // Pour imprimer l'idéal une fois sur deux.
int N;          //Le nombre d'éléments dans le poset.
```

```
int counter=0; // Le nombre d'idéaux imprimés.
int Y;         // Le deuxième minimum dans le poset courant.
typedef short Boolean;
typedef struct{
   int min;    // Le premier minimum dans le poset courant.
   int num;    // Le nombre d'éléments qui n'ont aucun élément plus petit qu'eux.
   unsigned int poset;
               // Chaque élément dans le poset est représenté par sa position
               // dans la représentation binaire du poset.
   int mask;   // Stocke les minima qui sont nécessaires pour calculer l'idéal suivant.
   int flag;   // Pour savoir dans quel cas nous sommes.
   } Item_type;
class Stack {
   private :
      int top; /* Sommet de la pile. */
      Item_type rgItem[MAXSTACK];
   public :
      Stack();
      Boolean EmptyStack();
      Boolean FullStack();
      int Push(Item_type item);
      int Pop(Item_type &item);
};
Stack stack;
Item_type item;
//-----------------------------------------------------
/*
 * Pile - initialise la pile.
 */
```

```
Stack : :Stack()
{
    top = 0;
}
//-----------------------------------------------------------
/*
 * Push - empile un élément, retourne 0 après succès, -1 sinon.
 */
int Stack : :Push(Item_type item) {
    if (top >= MAXSTACK)
        return(-1);
    else {
        rgItem[top++] = item;
        return(0);
    }
}
//-----------------------------------------------------------
/*
 * Pop - dépile un élément, retourne 0 après succès, -1 sinon.
 */
int Stack : :Pop(Item_type &item) {
    if (top <= 0)
        return(-1);
    else {
        item = rgItem[- -top];
        return(0);
    }
}
//-----------------------------------------------------------
```

```
/*
 * FullStack - la pile est-elle pleine ?.
 */
Boolean Stack : :FullStack() {
    return(top >= MAXSTACK);
}
//-----------------------------------------------------------
/*
 * EmptyStack - la pile est-elle vide ?.
 */
Boolean Stack : :EmptyStack() {
    return(top <= 0);
}
//-----------------------------------------------------------
/*
 * Retire le n-ième bit du mot.
 */
unsigned getbit(unsigned int word, int n) {
    return (word >> n) & 01;
}
//-----------------------------------------------------------
/*
 * Met le n-ième bit du mot à v.
 */
unsigned int setbit(unsigned int word, int n, unsigned v) {
    if (v != 0)
        return word | (01 << n);
    else
        return word & ~(01 << n);
```

```
}
//-----------------------------------------------------------
/*
 * Met à jour le tableau des relations et génère le tableau up
 * pour chaque élément du poset.
 */
void Process() {
    int i, j, k, count, bit;      // Réalise la transitivité.
    for (k=1; k <= N; k++){
        for (i=1; i <= N; i++){
            for (j=1; j <= N; j++){
                bit = getbit(table[i], j) | (getbit(table[i], k) & getbit(table[k], j));
                table[i] = setbit(table[i], j, bit);
            }
        }
    }
/* up[i] contient les éléments qui sont plus grands que i dans le poset.
    Par exemple, up[3] = 01101000 signifie que 3<5 et 3<6. */
    for (i=1; i <= N; i++){
        up[i] = 0;
        count = 0;
        for (j=1; j <= N; j++){
            if(getbit(table[i], j)){
                up[i] = setbit(up[i], j, 1);
            }
            if(getbit(table[j], i)){
                count++;
            }
        }
```

```
        steiner[i] = count - 1;
    }
}
//-----------------------------------------------------------
/*
 * Imprime l'idéal sous la forme {i, j, ··· }.
 */
void print(int number) {
    int i, flag = 0;
    if( !flip)
        flip = 1;
    else{
        flip = 0;
        printf("{");
        for(i=1; i <= N; i++)
            if(getbit(number, i)){
                if(flag)
                    printf(" %d", i);
                else
                    printf("%d", i);
                if( !flag)
                    flag = 1;
            }
            printf("}\ n");
            counter++;
    }
}
//-----------------------------------------------------------
/*
```

```
 * Trouve la position de 1 la plus à droite dans mask entre start et end.
 * Par exemple, search(000111, 1, 0, 5) = 2
 */
int search(unsigned int mask, int value, int start, int end){
    int pos;
    if(start > end){
        printf("index error\ n");
        exit(1);
    }
    if(start == end)
        return start;
    if(end == start + 1){
        if(getbit(mask, end) == value)
            return end;
        else if (getbit(mask, start) == value)
            return start;
        else{
            printf("\nfatal error!!!\ n");
            printf("mask is %d, start is %d, end is %d", mask, start, end);
            exit(1);}
    }
    else{
        pos = (start + end) / 2;
        if(getbit(mask, pos) == value)
            return (search(mask, value, pos, end));
        else
            return (search(mask, value, start, pos));
    }
}
```

```
//-------------------------------------------------------------
/*
 * Trouve l'élément minimum dans le poset.
 */
int findMinimal(unsigned long poset) {
    unsigned long res;
    int pos = 0;
    while(poset!=0){
        res = poset ^ (poset -1);    // Met chaque bit à 1, à partir du 1 le plus à droite jusqu'au
                                     // bit le plus à droite dans le poset, et met les autres à 0.
        pos = search(res, 1, pos, N);
        if(steiner[pos] == 0)
            break;
        poset = setbit(poset, pos, 0);
    }
    return pos;
}
//-------------------------------------------------------------
/*
 * Met à jour steiner[ ] et num après la suppression des minima,
 * afin de trouver le nouvel élément minimum.
 */
int Update(int poset, int min, int num) {
    long unsigned mask, res;
    int pos;
    mask = poset & up[min];
    pos = 0;
    while(mask!=0){
        res = (mask -1) ^ mask;
```

```
        pos = search(res, 1, pos, N);
        steiner[pos]- -;
        if(steiner[pos] == 0)
            num++;
        mask = setbit(mask, pos, 0);
    }
    return num -1;
}
//-------------------------------------------------------------
/*
 * Recover a l'effet inverse de Update, et en plus retourne num decrémenté par 1,
 * car après l'appel de Recover, nous supprimons la branche qui contient min
 * du poset.
 */
int Recover(int poset, int min, int num){
    long unsigned mask, res;
    int pos;
    mask = poset & up[min];
    pos = 0;
    while(mask != 0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);
        if(steiner[pos] == 0)
            num- -;
        steiner[pos]++;
        mask = setbit(mask, pos, 0);
    }
    return num;
}
```

```
//-------------------------------------------------------
/*
 * Retourne -1 si le poset est non linéaire (ordre total),
 * sinon retourne le nombre d'éléments dans le poset, excluant min.
 */
int linear(unsigned long poset, int min) {
    int number=1, i;
    for(i=min+1; i<=N; i++)
        if(getbit(poset, i)!=0)
        {
            if(steiner[i]==number)
                number++;
            else
                return -1;
        }
    return number-1;
}
//-------------------------------------------------------
/*
 * Si le poset est linéaire, imprime les idéaux sans utiliser la pile.
 */
void printLinear(unsigned long poset, int lin, int min, int mask){
    int resLin;
    if(lin == 0) // Le poset contient min seulement.
    {
        resLin = mask;
        resLin=setbit(resLin, min, 1);
        print(resLin);
        print(resLin);
```

```
    }
    else if(lin>0) // Le poset est totalement ordonné.
    {
        resLin = mask;
        for(int i=min; i<=N; i++)
            if(getbit(poset, i))
            {
                resLin=setbit(resLin, i, 1);
                print(resLin);
            }
        for( int i=N; i>=min; i--)
            if(getbit(poset, i))
            {
                print(resLin);
                resLin=setbit(resLin, i, 0);
            }
    }
}
//-------------------------------------------------------------------
/*
 * Génère les idéaux en utilisant la pile.
 */
void Ideal(unsigned long poset, Direction dir, int two_flag, int mask, int num) {
    int min, m, lin;
    unsigned int poset1, res;
    while( !stack.EmptyStack() || poset !=0){
        if(poset != 0){
            if(two_flag && num > 1)
                min = Y;
```

```
else
    min = findMinimal(poset);
if(num == 1){
    lin = linear(poset, min);
    if(lin>=0){
        printLinear(poset, lin, min, mask);
        poset=0;
    }
    else{ // poset n'est pas totalement ordonné.
        poset = setbit(poset, min, 0);
        num = Update(poset, min, num);
        mask = setbit(mask, min, 1);
        print(mask);
        two_flag = 0;
        item.min = min; item. num = num; item.poset = poset; item.mask = mask;
        item.flag =1;
        stack.Push(item);
    }
}
else{
    poset1 = poset;
    poset1 = setbit(poset1, min, 0);
    Y = findMinimal(poset1);
    res = setbit(mask, min, 1);
    if(dir == FORWARD){
        print(res);
        print(res);
        num = Update(poset1, min, num);
        item.min = min; item. num = num; item.poset = poset; item.mask = mask;
```

```
                item.flag =2;
                stack.Push(item);
                poset = poset1; dir = BACKWARD; mask = res; two_flag = 1;
            }
            else{
                item.min = min; item. num = num; item.poset = poset1; item.mask = res;
                item.flag =4;
                stack.Push(item);
                item.min = min; item. num = num; item.poset = poset1; item.mask = res;
                item.flag = 3;
                stack.Push(item);
                poset = (~up[min]) & poset;
                two_flag = 1; num=num-1;
                dir = BACKWARD;
            } // if(dir == FORWARD)
        } //if(num == 1)
    }
    else{ // (if poset != 0)
        stack.Pop(item);
        switch (item.flag){
            case 1 :
                print(item.mask);
                num = Recover(item.poset, item.min, item.num);
                break;
            case 2 :
                min = item.min; num = item.num; mask = item.mask; poset=item.poset;
                poset1 = poset;
                poset1 = setbit(poset1, min, 0);
                num = Recover(poset1, min, num);
```

```
                poset = (~up[min]) & poset;
                Y = findMinimal(poset);
                dir = FORWARD; two_flag = 1;
                break;
            case 3 :
                min = item.min; num = item.num; mask = item.mask; poset=item.poset;
                Y = findMinimal(poset);
                num = Update(poset, min, num);
                dir = FORWARD; two_flag = 1;
                break;
            case 4 :
                print(item.mask);
                print(item.mask);
                num = Recover(item.poset, item.min, item.num);
        }
    }
  }//while( !stack.EmptyStack());
}
//-------------------------------------------------------------
/*
 *
 */
int main(int argc, char* argv[ ]) {
  FILE *input;
  int i, left, right, num_zeros;
  unsigned long poset;
  char *fname;
  if(argc != 2){
      fprintf(stderr, "Usage : ideal in_file_name\n");
```

```
    exit(1);
}
fname = argv[1];
if((input = fopen(fname, "r")) == NULL){
    fprintf(stderr, "Error : could not read file %s\n", fname);
    exit(1);
}
//Donne le nombre d'éléments de l'ensemble.
fscanf(input, "%d" , &N);
printf("\n the number of elements in the poset is : %d\n\n", N);
for(i=1; i<=N; i++){
    table[i] = 0;
    table[i] = setbit(table[i], i, 1);
}
// Stocke l'ordre partiel.
int comma = 0;
printf(" the poset is \n { ");
do{
    fscanf(input, "%d %d", &left, &right);
    table[left] = setbit(table[left], right, 1);
    if(left)
        if(comma) printf(" , %d<%d", left, right);
        else printf("%d<%d", left, right);
    comma = 1;
}while(left != 0);
printf(" }\n");
fclose(input);
printf("\n\nIdeals in Gray Code Order\n");
// Initialise l'idéal et le poset.
```

```
    poset = 0;
    for(i=1; i <= N; i++)
        poset = setbit(poset, i, 1);
    Process();
    num_zeros = 0;
    for(i=1; i <= N; i++){
        if(steiner[i] == 0)
            num_zeros ++;
    }
    flip = 1;
    print(0);
    Ideal(poset, FORWARD, 0, 0, num_zeros);
    printf("%d ideals in total\n", counter);
    return 0;
}
```

RÉFÉRENCES

Aigner, M. 1979. *Combinatorial Theory*. Springer-Verlag, New York.

Amalraj, D.J., Sundararajan N. et Dhar G. 1990. « A data structure based on Gray code encoding for graphics and image processing », *Proceedings of the SPIE : International Society for Optical Engineering*, 65-76.

Ball, M.O. et Provan J.S. 1983. « Calculating bounds on reachability and connectedness in stochastic networks », *Networks*, 13 :253-278.

Beyer, T. et Ruskey F. 1989. « Constant average time generation of subtrees of bounded size », *Unpublished manuscript*.

Bitner, J.R., Ehrlich G. et Reingold E.M. 1976. « Efficient generation of the binary reflected Gray code and its applications », *Communications of the ACM*, 19(9) :517-521.

Bouchitte, V. et Habib M. 1989. « The calculation of invariants for ordered sets », *Algorithms et Order, I.Rival, ed., Kluwer Academic Publishing, Dordrecht*, 231-279.

Brightwell G. et Winkler P. 1992. « Counting Linear Extensions is #P-complete », *Order*, 8 :225-242.

Buck, M. et Wiedemann D. 1984. « Gray codes with restricted density », *Discrete Mathematics*, 48 :163-171.

Canfield, E.R. et Williamson S.G. 1995. « A loop-free algorithm for generating the linear extensions of a poset », *Order*, 12 :57-75.

Chang, C.C., Chen H.Y. et Chen C.Y. 1992. « Symbolic Gray codes as a data allocation scheme for two disc systems », *Computer Journal*, 35(3) :299-305.

Chase, P.J. 1970. « Algorithm 382 : Combinations of M out of N objects », *Communications of the ACM*, 13(6) :368.

———, P.J. 1989. « Combination generation and graylex ordering », *Proceedings of the 18th Manitoba Conference on Numerical Mathematics and Computing, Winnipeg*, Congressus Numerantium 69 :215-242.

Chen, M. et Shin K.G. 1990. « Sucube allocation and task migration in hyercube machines », *IEEE Transactions on Computers*, 39(9) :1146-1155.

Conway, J.H., Sloane N.J.A. et Wilks A.R. 1989. « Gray codes for reflection groups », *Graphs and Combinatorics*, 39(9) :1146-1155.

Cummings, R.L., 1966. « Hamilton circuits in tree graphs », *IEEE Trans. Circuit Theory*, 13 :82-90.

Diaconis, P. et Holmes S. 1994. « Gray codes for randomization procedures », *Statistics and Computing*, 4 :287-302.

Dietz, P.F. 1989. « Optimal algorithms for list indexing and subset rank », *Proceedings of the 1989 Workshop on Algorithms and Data Structures in Ottawa, Canada*, Lecture Notes in Computer Science 382, Springer-Verlag, Berlin, 39-46.

Eades, P., Hickey B. et Read R.C., 1984. « Some Hamilton paths and a minimal change algorithm », *Journal of the ACM*, 31(1) :19-29.

Eades, P. et McKay B. 1984. « An algorithm for generating subsets of fixed size with a strong minimal change property », *Information Processing Letters*, 19 :131-133.

Ehrlich, G. 1973. « Loopless algorithms for generating permutations, combinations, and other combinatorial configurations », *Journal of the ACM*, 20(1) :500-513.

Faloutsos, C. 1988. « Gray codes for partial match and range queries », *IEEE Transactons on Software Engineering*, 14(10) :1381-1393.

Foundas, E. 1991. « Algorithms on involutions », *Journal of Information and Optimization Sciences*, 12 :339-361.

Gardner, Martin 1972. « Curious properties of the Gray code and how it can be used to solve puzzles », *Scientific American*, 227(2) :106-109.

Garey, M.R. et Johnson D.S. 1979. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Co.

Gilbert, E.N. 1958. « Gray codes and paths on the n-cube », *Bell Systems Technical Journal*, 37 :815-826.

Gray, F. March 1953. « Pulse code communications », *U.S. Patent*, 2632058.

Heath, F.G. 1972. « Origins on the binary code », *Scientific American*, 227(2) :76-83.

Holzmann, C.A. et Harary F. 1972.« On the tree graph of a matroid », *SIAM J. Appl. Math.*, 22(2) :187-193.

Hu, T.C. et Ruskey F. 1980. « Circular cuts in a network », *Mathematics of Operations Research*, 5 :422-434.

Johnson, S.M. 1963. « Generating of permutations by adjacent transpositions », *Math. Comp.* , 17 :282-285.

Joichi, J.T., White Dennis E. et Williamson S.G. 1980. « Combinatorial Gray codes », *SIAM Journal on Computing*, 9(1) :130-141.

Kalvin, A.D. et Varol Y.L. 1983. « On the generation of all topological sortings », *J. Algorithms*, 4 :150-162.

Knuth, D.E. 1973. *The art of computer programming.* Vol. 3 (sorting and searching), Addision-Wesley, Reading, Mass., 578-579.

Knuth, D.E. et Szwarcfiter J. 1974. « A structured program to generate all topological sorting arrangements », *Inform. Process. Lett.*, 3 :153-157.

Koda, Y. et Ruskey F. 1993. « A Gray Code for the ideals of a forest poset », *J. Algorithms*, 15 :324-340.

Korsh, J.F. et Lafollette P.S. 2002. « Loopless Generation of Linear Extensions of a Poset », *Order*, 19 :115-126.

Lawler, E.L. 1979. « Efficient implementation of dynamic programming algorithms for sequencing problems », *Stichting Mathematisch Centrum*, Technical Report BW106/79.

Lehmer, D.H. 1965. « Permutation by adjacent interchanges », *American Mathematical Monthly*, 72 :36-36.

Liu, C.N. et Tang D.T. 1973. « Algorithm 452, Enumerating M out of N objects », *Comm. ACM*, 16 :485.

Losee, R.M. « A Gray code based ordering for documents on shelves : Classification for browsing and retrieval », *Journal of the American Society for Information Science*, 43(4) :312-322.

Lucas, J. 1987. « The rotation graph of binary trees is hamiltonian », *Journal of Algorithms*, 8 :503-585.

Lucas, J.M., Roelants van Baronaigien D. et Ruskey F.1993. « On rotations and the generation of binary trees », *Journal of Algorithms*, 15(3) :343-366.

Ludman, J.E. 1981. « Gray code generation for MPSK signals », *IEEE Transactions on Communications*, COM-29 :1519-1522.

Nijenhuis, A. et Wilf H.S. 1978. *Combinatorial Algorithms for Computers and Calculators.* Academic Press.

Proskurowski, A. et Ruskey F. 1990. « Generating binary trees by transpositions », *J. Algorithms*, 11 :68-84.

Pruesse, G. et Ruskey F. 1991. « Generating the linear extensions of certain posets by adjacent transpositions », *SIAM Journal of Discrete Mathematics*, 4 :413-422.

———. 1993. « Gray Codes from Antimatroids », *Order*, 10 :239-252.

———. 1994. « Generating linear extensions fast », *SIAM Journal on Computing*, 23 :373-386.

Rao, G.S., Stone H.S. et Hu T.C., « Scheduling in a Distributed Processor System with Limited Memory », *IEEE Trans. Computers*, C-28 :291-299.

Reingold, Nievergelt et Deo 1977. *Combinatorial Algorithms : Theory and Practice*. Prentice Hall.

Richard, D. 1986. « Data compression and Gray-code sorting », *Information Processing Letters*, 22 :210-215.

Robinson, J. et Cohn M. 1981.« Counting sequences », *IEEE Transactions on Computers*, C-30 :17-23.

Roelants van Baronaigien, D. 1992. « Constant time generation of involutions », *Congressus Numerantium*, 90 :87-96.

Ruskey, F. 1978. *Algorithmic Solution of Two Combinatorial Problems*. Information and Computer Science, U.C. San Diego, 40-42.

———. 1981. « Listing and Counting Subtrees of a Tree », *Computing*, 10 :141-150.

———. 1988. « Adjacent interchange generation of combinations », *Journal of Algorithms*, 9 :162-180.

———. 1992. « Generating linear extensions of posets by transpositions », *Journal of Combinatorial Theory Series B*, 54 :77-101.

———. 1993. « Simple combinatorial Gray codes constructed by reversing sublists », *L.N.C.S*, 762 :201-208.

Savage, C.D. 1989. « Gray code sequences of partitions », *Journal of Algorithms*, 10(4) :577-595.

———. 1997. « A survey of combinatorial Gray codes », *SIAM*, 39(4) :605-629.

Schrage, L. et Baker K.R. 1987. « Dynamic programming solution of sequencing problems with precedence constraints », *Operations Research*, 26(3) :444-449.

Squire, Matthew. « Enumerating the Ideals of a Poset », *http ://cite seer.ist.psu.edu/465417.html*.

Stachowiak, G. 1992. « Hamilton paths in graphs of linear extensions for unions of posets », *SIAM Journal on Discrete Mathematics*, 5 :199-206.

Stanley, R.P. 1986. *Enumerative Combinatorics*. Wadsworth, Belmont, CA., Vol.1.

Steiner, G. 1986. « An algorithm to generate the ideals of a partial order », *Operatons Research Letters*, 5(6) :317-320.

Steinhaus, H. 1963. *One Hundred Problems in Elementary Mathematics*. Pergamon Press, New York and Oxford.

Trotter, H.F. 1962. « PERM (Algorithm 115) », *Communications of the ACM*, 5(8) :434 -435.

Vajnovszki, V. et Walsh T.R. 2003. « A loopless two-close Gray code algorithm for listing k-aray Dyck words », *Rapport de recherche, Département d'informatique, UQÀM*, No 03-01. À apparaître dans *Journal of Discrete Algorithms*.

Varol, Y.L. et Rotem D. 1981. « An algorithm to generate all topological sorting arrangements », *Computer J.*, 24 :83-84.

Walsh, T.R. 1995. « A simple sequencing and ranking method that works on almost all Gray codes », *Department of Mathematics and Computer Science, Université du Québec à Montréal*, Research report 243, 53 pages.

———. 1998. « Generation of Well-Formed Parenthesis Strings in Constant Worst-Case Time », *Journal of Algorithms*, 29 :165-173.

———. 2000. « Loop-free sequencing of bounded integer compositions », *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 33 :323-345.

———. 2001. « Gray Codes for involutions », *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 36 :95-118.

Wells, M.B. 1961. « Generation of permutations by transposition », *Mathematics of Computation*, 15 :192-195.

———. 1971. *The elements of combinatorial computing*. Pergamon Perss, New York and Oxford.

West, D.B. 1993. « Generating linear extensions by adjacent transpositions », *Journal of Combinatorial Theory Series B*, 57 :58-64.

Wilf, H.S. 1989. « Combinatorial Algorithms : An Update », *SIAM, CBMS 55*.

Williamson, S.G. 1985. *Combinatorics for computer science*. Computer Science Press, Rockville.