UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UN NOUVEL ALGORITHME POUR LE CALCUL DES GÉNÉRATEURS DES INTERVALLES COMMUNS DE K PERMUTATIONS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ANDRÉ LEVASSEUR

DÉCEMBRE 2006

REMERCIEMENTS

Je tiens à remercier ma directrice de recherche Anne Bergeron qui, par son expérience, sa disponibilité et sa générosité, m'a permis de réaliser un tel travail. J'ai beaucoup apprécié ses mots encourageants et la confiance dont elle m'a fait preuve en me proposant ce projet, de son propre aveu, plutôt difficile.

Je remercie aussi ma conjointe, Jacqueline Rwirangira, qui pendant cette période a non seulement donné naissance à notre merveilleuse petite Maya, mais a aussi donné tant de temps, sans compter. Son amour et son soutien ont été essentiels, en particulier dans les difficiles moments de nouveaux parents.

Un tel projet ne peut se faire sans financement et à ce chapitre, je remercie une nouvelle fois ma directrice Anne Bergeron, les différents programmes de bourses de la Fondation UQAM et le CRSNG.

En terminant, je souhaite remercier mes défunts parents qui ont su m'inculquer la satisfaction de solutionner des problèmes avec ingéniosité et l'ouverture d'esprit nécessaire à la science.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

DÉCLARATION DES VARIABLES

a Utilisé dans 2 exemples : position la plus à gauche où un élément de l'ensemble $\{i, i+1, \ldots, j\}$ est rencontré dans P

A Un ensemble ou un intervalle quelconque

b Utilisé dans 2 exemples : position la plus à droite où un élément de l'ensemble $\{i, i+1, \ldots, j\}$ est rencontré dans P

B Un ensemble quelconque ou un intervalle quelconque

c Un intervalle commun de $\mathcal{P}$ (Heber et Stoye)

$C_{\mathcal{P}}$ Ensemble des intervalles communs de $\mathcal{P}$ (Heber et Stoye)

$\mathcal{C}$ Classe de chevauchement

F Un intervalle commun fort

$\mathcal{G}(R, L)$ La fermeture transitive de la relation de chevauchement sur $R \cup L$

i Itérateur général en terme de position ou d'élément (ex $P[i]$, $R[i]$, $Sup[i]$, P_i, p_i, $I(i)$) ou élément extrémal d'un intervalle $(i..j)$

i^* Pour un i donné, position telle que p_{i*} est l'élément plus petit que p_i qui est le plus près, à gauche de p_i

$(i..j)$ Intervalle de la permutation identité allant de l'élément i à j

$I(i)$ Dans l'ensemble d'intervalles I, intervalle pour l'élément i

Id_n La permutation identité comprenant n éléments

$IMax$ Vecteur d'intervalles où $IMax[p_i]$ est le plus grand intervalle de P où chacun de ses éléments est $\geq p_i$

$IMin$ Vecteur d'intervalles où $IMin[p_i]$ est le plus grand intervalle de P où chacun de ses éléments est $\leq p_i$

Inf Vecteur du générateur (Sup, Inf)

j Itérateur général en terme d'élément (ex $L[j]$, $Inf[j]$) ou élément extrémal d'un intervalle $(i..j)$

$J(i)$ Dans l'ensemble d'intervalles J, intervalle pour l'élément i

k Élément quelconque d'une permutation (sert d'itérateur)

K Nombre de permutations de l'ensemble $\mathcal{P}$

L Vecteur du générateur (R, L)

$L[\mathcal{C}]$ Borne gauche de la classe de chevauchement $\mathcal{C}$

M Liste d'intervalles communs (Heber et Stoye)

n Taille d'une permutation

N Nombre d'intervalles communs d'un ensemble $\mathcal{P}$

N_i Nombre d'intervalles communs entre P_1 et P_i (Heber et Stoye)

p_i Élément situé à la position i d'une permutation P

P Une permutation sur n éléments

$P[i]$ Synonyme de p_i

$\mathcal{P}$ Ensemble de K permutations

$\mathcal{P}_i$ Une permutation P de l'ensemble $\mathcal{P}$

$\mathcal{P}'$ Ensemble réétiqueté de K permutations pour inclure Id_n

Q Une autre permutation sur n éléments

$\mathcal{Q}$ Ensemble de permutations

R Vecteur du générateur (R, L)

$R[\mathcal{C}]$ Borne droite de la classe de chevauchement $\mathcal{C}$

s L'élément au dessus d'une pile

S Une pile ('S' pour "Stack")

Sup Vecteur du générateur (Sup, Inf)

X Vecteur pour expliquer la notation $min(X, Y)$

Y Vecteur pour expliquer la notation $min(X, Y)$

Π Opération de projection

RÉSUMÉ

Le but principal de ce travail est d'apporter une synthèse des travaux effectués sur les algorithmes capables d'identifier des groupes de gènes, ou des segments de génomes, communs à différentes espèces. Nous nous limitons à une approche basée sur la modélisation des génomes à l'aide de permutations, où les groupes d'éléments conservés entre différents génomes sont définis par la notion d'intervalles communs. L'étude approfondie de cette problématique nous a aussi permis d'innover et de présenter un nouvel algorithme plus général.

Ce travail comporte donc deux aspects principaux, bien que le second soit intercalé dans le premier : une partie synthèse et une partie innovatrice. Après un survol des notions de bases, la partie synthèse présente les premiers algorithmes qui permettent de calculer les intervalles communs, puis l'approche de Bergeron et al. (2005) qui a grandement simplifié les premières solutions. Cette partie comprend aussi les algorithmes des mêmes auteurs qui calculent les intervalles communs dits forts, et une discussion sur les avantages de leur utilisation par rapport aux intervalles communs classiques. La partie innovatrice présente un nouvel algorithme qui généralise deux algorithmes de Bergeron et al. (2005). Nous concluons cette étude en constatant l'extrême simplicité et la grande efficacité, autant théorique que pratique, des plus récents algorithmes et nous croyons peu vraisemblable qu'il soit possible d'en améliorer la performance de façon significative.

MOTS CLÉS : algorithmique, génomique comparée, intervalles communs, permutations.

INTRODUCTION

Les êtres vivants sont divisés en deux classes majeures : les procaryotes comme par exemple les bactéries et les eucaryotes qui regroupent des organismes plus complexes comme les plantes ou les mammifères. Dans tous les cas, l'information génétique est contenue dans une ou plusieurs très longues molécules d'ADN qui constituent le génome d'un individu. L'ADN est typiquement représenté par une séquence de lettres sur l'alphabet A, C, G, T où chaque lettre représente une base. Certains segments de cette séquence sont capables de produire des protéines et correspondent alors à des gènes. Grâce à la technique du séquençage, il est possible de déterminer la séquence des bases d'une molécule d'ADN ce qui permet leur comparaison entre différents organismes. On dira de deux séquences ou de deux génomes qui se ressemblent, que leur distance est faible. Ces mesures de distances sont utilisées en phylogénétique moléculaire afin de déterminer la hiérarchie des relations d'évolution de différentes espèces. On représente cette hiérarchie à l'aide d'arbres où les feuilles sont les espèces actuelles et les nœuds internes leurs ancêtres.

La quantité de données biologiques disponibles croit de façon exponentielle (voir Figure 0.1). Entre août 1997 et août 2005, son volume a crû de 100 fois (ce qui correspond en moyenne à doubler à tous les 14 mois), pour atteindre 100 milliards de paires de bases. De plus, grâce aux améliorations des techniques de séquençage, on s'attend à ce que cette croissance explosive se poursuive (NCBI News 2005). En termes de gènes, en mai 2006, le moteur de recherche Entrez Gene du NCBI comptait 1.9 millions de gènes répartis dans 3315 génomes. De ce nombre, il y avait 362 génomes dont le nombre de gènes disponibles était supérieur à 1000 pour une moyenne de 4841 gènes par génome. Cette moyenne varie grandement selon le règne considéré (voir Tableau 0.1). Elle est supérieure à dix mille gènes chez les eucaryotes et dépasse allègrement les vingt mille chez les mammifères (voir

Tableau 0.2).

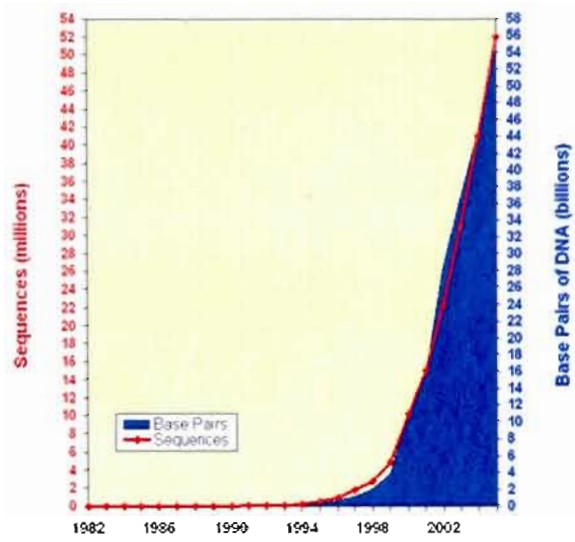


FIG. 0.1 Croissance exponentielle des données biologiques disponibles. Source : NCBI.

TAB. 0.1 Nombre moyen de gènes selon le règne parmi les génomes dont au moins 1000 gènes sont disponibles.

Règne	Nb de gènes moyen	Nb de génomes
Eukaryotes	13482	51
Eubacteries	3481	282
Archéobactéries	2503	26

Données compilées d'après les données de Entrez Gene du NCBI (mai 2006)

La notion d'intervalles communs a été introduite par Uno et Yagiura (2000) afin de modéliser le fait qu'un même groupe de gènes peut se retrouver dans un ordre différent dans les génomes de différentes espèces. Cette conservation de voisinage sert en biologie à identifier l'existence d'un groupe fonctionnel de protéines qui interagissent. L'ordre des gènes d'un génome est modélisé par une permutation où chaque gène est étiqueté par un entier de 1 à n.

TAB. 0.2 **Nombre de gènes disponibles selon le mammifère**

Espèce	Nb de gènes
Chien	20152
Chimpanzé	24153
Homme	39256
Rat	28569
Souris	60549
Taureau	23655

Données obtenues de Entrez Gene du NCBI (mai 2006)

Les espèces phylogénétiquement proches ont des génomes très semblables. Par exemple, les nucléotides des séquences de l'homme et du chimpanzé ne divergent que d'environ 1% (Marquès-Bonet et al, 2004), (The Chimpanzee Sequencing and Analysis Consortium, 2005). Les permutations obtenues de la modélisation de ces génomes sont alors très semblables, ce qui nous mène à un nombre $O(n^2)$ d'intervalles communs. Pour des espèces voisines, cela correspond, en moyenne, à plus de 10 millions d'intervalles communs chez les eubactéries, à près de 200 millions chez les mammifères et à environ 1.6 milliards entre l'homme et le chimpanzé.

En termes de complexité temporelle, les meilleures solutions obtenues jusqu'ici pour calculer les intervalles communs d'un ensemble de K permutations sur n éléments sont $O(Kn + N)$, où N est le nombre d'intervalles communs. On considère cette complexité optimale puisque proportionnelle à la taille des entrées et sorties. Cependant, étant donné la forte croissance du nombre de génomes disponibles (K) et surtout la grande taille de N dans le cas d'espèces voisines, il s'avère important d'avoir un algorithme efficace en pratique, simple et facilement vérifiable.

La solution de Hèber et Stoye (2001) est basée sur un algorithme compliqué et utilise une lourde structure de données. Ces deux facteurs la rendent non seulement peu efficace mais aussi difficile à implémenter. Les travaux de Bergeron et al (2005) ont

permis de simplifier grandement à la fois l'algorithme et sa structure de données, ce qui a conduit à une solution très efficace et facile à implanter, le tout dans un espace $O(n)$. De plus, cette solution introduit la notion des générateurs qui permet une représentation des $O(n^2)$ intervalles communs dans un espace linéaire, ce qui constitue un important gain par rapport à une représentation exhaustive. L'aspect innovateur du présent travail réside dans la présentation d'un nouvel algorithme qui généralise le calcul de ces générateurs.

La recherche de groupes conservés à l'aide d'intervalles communs est cependant trop sensible. Une grande part des intervalles rapportés ont une faible valeur informative et à l'opposé, les intervalles conservés introduits par (Bergeron et Stoye, 2003) ne sont pas assez nombreux. Les intervalles communs forts (F. de Montgolfier, 2003) permettent non seulement d'identifier tous les groupes conservés significatifs, mais aussi une représentation visuelle directe grâce à leur nombre linéaire.

Le chapitre I explique les notions de base reliées aux intervalles communs puis présente un survol des premiers algorithmes précédant les travaux de Bergeron et al. (2005). Le chapitre II décrit en détails la notion des générateurs de ces derniers auteurs. Le chapitre III présente d'abord la méthode de calcul d'un générateur, telle que proposée par ses auteurs, puis une nouvelle méthode plus générale, basée sur une opération que l'on nomme *projection*. Nous terminons ce chapitre en montrant comment, à partir d'un générateur, on obtient aisément les intervalles communs d'un ensemble de K permutations. Le dernier chapitre introduit un type particulier de générateur avec lequel nous montrons comment calculer les intervalles communs forts, et pour clore le présent travail, nous discutons de la pertinence de ces intervalles et de leur représentation à l'aide d'un arbre.

CHAPITRE I

LES INTERVALLES COMMUNS ET LES PREMIERS ALGORITHMES

Nous présentons d'abord un rappel des notions de base soit les permutations, les intervalles et les intervalles communs puis nous discutons des travaux de Yagiura, Nagamochi et Ibaraki (1995), d'Uno et Yagiura (2000) et d'Heber et Stoye (2001), qui ont produit en tout cinq algorithmes permettant chacun de calculer des intervalles communs.

1.1 Les permutations

Une permutation P sur n éléments est un ordre linéaire sur l'ensemble d'entiers $\{1, 2, \ldots, n\}$. On note Id_n la permutation $(1, 2, \ldots, n)$. Une permutation n'admet pas d'éléments répétés et doit contenir chacun des n éléments de l'ensemble de référence.

1.2 Les intervalles

Un intervalle d'une permutation $P = (p_1, p_2, \ldots, p_n)$ sur n éléments est un ensemble d'éléments consécutifs de la permutation P. Sauf avis contraire, la borne gauche d'un intervalle est l'indice dans P du premier élément de cet intervalle et la borne droite est l'indice du dernier élément. Sinon, lorsque spécifié, la borne sera non pas la position mais l'élément qui occupe cette position. On identifie un intervalle soit par l'ensemble des éléments qui le composent, ou en donnant sa borne gauche et sa borne droite. Les deux bornes sont notées entre crochets si l'intervalle est sur une permutation quelconque

et entre parenthèses si l'intervalle est un intervalle de la permutation identité.

Par exemple, soit $P = (1\ 3\ 2\ 5\ 4)$ et l'intervalle composé des éléments $\{2, 3, 5\}$. La position 2 qu'occupe l'élément 3 constitue la borne gauche de cet intervalle et la position 4 qu'occupe l'élément 5 en constitue la borne droite. On note alors cet intervalle [2..4]. L'intervalle $\{1, 2, 3\}$ peut être identifié par (1..3) car les positions 1 à 3 correspondent aux éléments 1, 2 et 3 qui constituent un intervalle sur Id_n. Par contre, les éléments $\{2, 3, 4\}$ ne forment pas un intervalle puisqu'ils ne sont pas consécutifs dans P.

1.3 Les intervalles communs

1.3.1 Notions de base

Définition 1. Soit $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ un ensemble de K permutations sur n éléments. Un *intervalle commun* de $\mathcal{P}$ est un ensemble d'entiers qui est un intervalle pour chacune des permutations de $\mathcal{P}$.

Par exemple, pour $\mathcal{P} = \{$ (**1 2 8 7** 5 6 4 3), (6 3 **7 8 1 2** 4 5), (6 5 4 3 **2 1 7 8**), (6 **8 2 1 7** 5 4 3)$\}$, l'ensemble des éléments $\{1, 2, 7, 8\}$ constitue un intervalle commun de $\mathcal{P}$. Ce n'est pas le cas pour l'ensemble des éléments $\{1, 2, 3\}$.

Remarque 1. L'ensemble $\{1, 2, \ldots, n\}$ et chacun des singletons sont des intervalles communs de tout ensemble non vide de permutations. Ces intervalles sont dits *triviaux*. Par exemple, pour tout ensemble $\mathcal{P}$ de permutations sur 7 éléments, les intervalles communs triviaux sont {1}, {2}, {3}, {4}, {5}, {6}, {7}, {1, 2, 3, 4, 5, 6, 7}.

Dans la suite du présent travail nous supposerons, sans perte de généralité, que l'ensemble $\mathcal{P}$ contient la permutation identité Id_n. Il n'y a pas de perte de généralité parce qu'il est toujours possible de réétiqueter les différentes permutations de $\mathcal{P}$ de façon à obtenir une permutation identité. Évidemment, le changement d'étiquette n'affecte en

rien l'essence même des permutations, soit l'ordre de leurs éléments. Par exemple,

$$\mathcal{P} = \{(2,4,3,5,1),(2,5,3,4,1),(5,4,3,2,1)\}$$

qui ne contient pas Id_5, peut se réécrire

$$\mathcal{P}' = \{(1,2,3,4,5),(1,4,3,2,5),(4,2,3,1,5)\}.$$

Il a suffit d'établir une correspondance entre chaque élément de la permutation $(2,4,3,5,1)$ de $\mathcal{P}$ et l'élément situé à la même position dans la permutation identité $(1,2,3,4,5)$. On obtient alors une table de traduction qu'on applique ensuite à chaque permutation de $\mathcal{P}$ pour obtenir les permutations de $\mathcal{P}'$. Cet exemple réétiquette la permutation $(2,4,3,5,1)$ vers l'identité mais on aurait pu aussi bien utiliser n'importe quelle autre permutation de $\mathcal{P}$ vers l'identité.

La présence obligatoire de Id_n dans l'ensemble $\mathcal{P}$ permet de représenter tout intervalle commun en se basant sur cette permutation identité, où chaque élément correspond à la position qu'il occupe. Ainsi, la notation d'un intervalle commun $(i..j)$, $i \leq j$, indique directement quels éléments en font partie, puisque i et j sont à la fois positions et éléments.

Par exemple, pour $\mathcal{P} = \{(1,2,3,4,5),(5,3,2,4,1)\}$, l'intervalle commun $(2..5)$ contient forcément les éléments 2, 3, 4 et 5 car, dans la permutation identité, les positions 2 et 5 correspondent respectivement aux éléments 2 et 5 et, qu'entre ces éléments, il y a nécessairement tous les entiers plus grands que 2 et plus petits que 5. Par contre, si on a $\mathcal{Q} = \{(2,4,1,5,3),(5,1,4,2,3)\}$ qui ne contient pas Id_n, l'intervalle commun $\{1,2,4\}$ ne peut s'exprimer qu'en énumérant ses éléments. Il correspond à l'intervalle [1..3] dans la première permutation et à [2..4] dans la deuxième. La notation par crochets ne permet pas de savoir directement que l'élément 4 fait partie de ces deux derniers intervalles et la notation parenthésée ne peut être utilisée sans Id_n.

1.3.2 Intervalles communs chevauchants

Nous voyons ici quelques propriétés fondamentales des intervalles communs qui s'avéreront fort utiles pour la définition des intervalles irréductibles (voir Section 1.5.1) de Heber et Stoye (2001), ou lors de démonstrations touchant les générateurs canoniques (Section 4.1) et les intervalles forts (Section 4.2).

Définition 2. Deux intervalles communs A et B se *chevauchent* si $A \cap B \neq \varnothing$ et si l'un n'est pas inclus dans l'autre.

Proposition 1. *Soit A et B deux intervalles communs chevauchants d'un ensemble de permutations $\mathcal{P}$, alors $A \cup B$, $A \cap B$, $A \setminus B$ et $B \setminus A$ sont aussi des intervalles communs de $\mathcal{P}$.*

Démonstration. Rappelons ici que l'on peut supposer, sans perte de généralité, que la permutation identité fait partie de $\mathcal{P}$. Soit i et j étant respectivement le plus petit et le plus grand élément d'un intervalle. Alors les éléments k tel que $i \leq k \leq j$ composeront cet intervalle si et seulement si cet intervalle est commun.

Si $C = A \cap B$ n'est pas un intervalle commun, alors pour les deux entiers i et j de C, les éléments k, $i \leq k \leq j$, tels que $k \notin C$ seront nécessairement situés dans les segments $A \setminus B$ et $B \setminus A$, puisque A et B sont des intervalles communs. Puisque les éléments k sont dans A et B, ils font partie de l'intersection C, ce qui contredit la définition de k.

Puisque $A \cap B$ et A sont des intervalles communs, directement, $A \setminus B$ est aussi un intervalle commun. De plus, les éléments de l'intervalle $A \setminus B$ allongent l'intervalle $A \cap B$, soit vers le haut (les éléments de $A \setminus B$ sont tous supérieurs et immédiatement consécutifs à ceux de $A \cap B$) ou soit vers le bas (les éléments de $A \setminus B$ sont tous inférieurs et immédiatement consécutifs à ceux de $A \cap B$). Similairement, si $A \cap B$ et B sont des intervalles communs, $B \setminus A$ est aussi un intervalle commun.

Si A, B, $A \cap B$, $A \setminus B$ et $B \setminus A$ sont des intervalles communs de $\mathcal{P}$, deux cas sont

alors possibles :

1. $A \setminus B$ allonge l'intervalle $A \cap B$ vers le bas alors $B \setminus A$ allonge l'intervalle $A \cap B$ vers le haut et $A \cup B$ est un intervalle commun.
2. $A \setminus B$ allonge l'intervalle $A \cap B$ vers le haut alors $B \setminus A$ allonge l'intervalle $A \cap B$ vers le bas et $A \cup B$ est un intervalle commun.

□

La Figure 1.1 aide à visualiser ces propriétés lorsque deux intervalles communs se chevauchent.

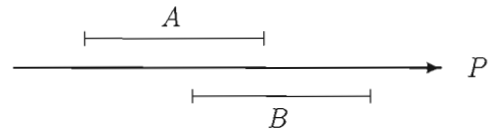


FIG. 1.1 Le chevauchement des intervalles communs A et B sur P implique que $A \cup B$, $A \cap B$, $A \setminus B$ et $B \setminus A$ sont aussi des intervalles communs (voir la démonstration de la Proposition 1).

1.4 Les premiers algorithmes

Un premier algorithme de calcul des intervalles communs a d'abord été publié en japonais par Yagiura, Nagamochi et Ibaraki (1995) et puis trois autres, en anglais, par Uno et Yagiura (2000). Chacun de ces algorithmes calcule l'ensemble des intervalles communs de deux permutations, sans calculer les intervalles de taille 1.

Le premier algorithme, nommé *basic algorithm* par ses auteurs, en est un de force brute qui examine tous les intervalles possibles. Sa complexité temporelle est dans tous les cas de $O(n^2)$. Les deux suivants, LHP et MNG, sont de simples améliorations du premier où des tests sont ajoutés pour éviter certains calculs redondants, sans toutefois en réduire la complexité théorique. En pratique cependant, ils sont en général beaucoup plus rapides que l'algorithme de base. Le dernier algorithme, appelé *Reduce Candidate*, s'effectue en temps $O(n + N)$, où n est la taille d'une permutation et N le nombre

d'intervalles communs. Cependant, N est dans $O(n^2)$ ce qui implique, dans le pire cas, une complexité temporelle de $O(n^2)$, mais néanmoins optimale, puisque proportionnelle à la taille des entrées et sorties.

Bien que plus intéressant du point de vue théorique, l'algorithme *Reduce Candidate* souffre de trois inconvénients importants : la difficulté d'en comprendre les propriétés (Xuan, Habib et Paul, 2005), son implémentation ardue, et sa lenteur d'exécution. De leur propre aveu, ses auteurs le qualifient de *"très compliqué"* et rapportent, qu'en pratique, les algorithmes $O(n^2)$ plus simples, LHP et MNG, sont beaucoup plus rapides. Cependant, dans certains cas, ces deux derniers demandent un temps $\Omega(n^2)$. Ainsi Uno et Yagiura (2000) ne recommandent *Reduce Candidate* que pour des permutations de grande taille ($n > 10^5$) et si on a beaucoup de temps à consacrer à sa programmation.

1.5 Une extension pour K permutations

Afin de pouvoir calculer les intervalles communs d'un ensemble de K permutations, Heber et Stoye (2001) ont ajouté une extension non triviale à l'algorithme *Reduce Candidate* d'Uno et Yagiura (2000) où K, rappelons-le, était nécessairement égal à 2.

Pour un ensemble $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$, une approche naïve aurait été d'utiliser répétitivement *Reduce Candidate* pour identifier les intervalles communs de P_1 et P_i pour $2 \leq i \leq K$ desquels on ne conserverait que ceux qui sont communs aux K permutations. La complexité temporelle d'un tel algorithme serait $O(Kn + \sum_{i=2}^{K} N_i)$ où N_i est le nombre d'intervalles communs entre P_1 et P_i pour $2 \leq i \leq K$.

Plus efficace, l'algorithme d'Heber et Stoye (2001) permet d'obtenir une complexité temporelle de $O(Kn + N)$. Leur approche repose sur une représentation de l'ensemble $C_{\mathcal{P}}$ de tous les intervalles communs de $\mathcal{P}$ par un sous ensemble $I_{\mathcal{P}}$ plus petit d'intervalles, dits *irréductibles*, à partir duquel $C_{\mathcal{P}}$ peut-être facilement calculé.

On définit ci-dessous les intervalles irréductibles puis on décrit succinctement l'algorithme.

1.5.1 Les intervalles irréductibles

Soit $C_{\mathcal{P}}$ l'ensemble des intervalles communs d'un ensemble $\mathcal{P}$ de K permutations. Une liste $M = (c_1, \ldots, c_{l(M)})$ d'intervalles communs $c_1, \ldots, c_{l(M)} \in C_{\mathcal{P}}$ est une *chaîne*, de taille $l(M)$, si chaque paire d'intervalles consécutifs de M se chevauchent. Une chaîne de taille 1 est dite *chaîne triviale* et toutes les autres sont des *chaînes non triviales*. Puisque l'union de deux intervalles communs chevauchants est aussi un intervalle commun (voir Proposition 1), chaque chaîne M génère un intervalle commun $c = \bigcup_{c' \in M} c'$. Un intervalle commun est *réductible* s'il existe une chaîne non triviale qui le génère, sinon il est *irréductible*.

Heber et Stoye (2001) ont démontré que le nombre d'intervalles irréductibles $|I_{\mathcal{P}}|$ est inférieur ou égal à n, soit nettement moins que l'espace quadratique de $|C_{\mathcal{P}}|$, le nombre d'intervalles communs de l'ensemble $\mathcal{P}$.

1.5.2 L'algorithme de Heber et Stoye

Heber et Stoye (2001) ont modifié l'algorithme *Reduce Candidate* de Uno et Yagiura (2000) de façon à ce qu'il calcule $I_{\mathcal{P}_i}$, l'ensemble des intervalles communs irréductibles des permutations Id_n et $\mathcal{P}_i$, en un temps $O(n)$. Ce calcul est répété itérativement pour i allant de 2 à K et à la fin, on obtient $I_{\mathcal{P}}$, i.e. l'ensemble des intervalles communs irréductibles des K permutations de $\mathcal{P}$ en un temps $O(Kn)$.

La structure de données de *Reduce Candidate*, trois listes doublement chaînées, a été augmentée par plusieurs autres listes du même type, une pour chaque chaîne maximale de $\mathcal{P}_{i-1}$. De plus, des pointeurs verticaux relient certains noeuds de ces listes pour former des listes verticales. Cette lourde structure, qui est en plus mise à jour à chaque itération, occupe malgré tout un espace linéaire en fonction de n.

À partir de l'ensemble d'intervalles irréductibles $I_{\mathcal{P}}$ obtenu, une dernière procédure calcule l'ensemble $C_{\mathcal{P}}$ des intervalles communs de l'ensemble $\mathcal{P}$ de permutations en un temps proportionnel au nombre N d'intervalles communs. On obtient ainsi une

complexité temporelle $O(Kn+N)$ pour la procédure complète. Ce temps est asymptotiquement optimal puisque proportionnel à la taille des entrées et sorties. Cependant, d'un point de vue pratique, l'ajout d'une structure de données complexe et sa mise à jour dynamique ajoutent encore plus de lourdeur à l'algorithme *Reduce Candidate*, déjà reconnu difficile et peu performant en pratique. Il aurait été intéressant de comparer la performance d'une utilisation naïve de LHP et MNG d'Uno et Yagiura (2000) avec l'extension d'Heber et Stoye (2001) mais ces derniers n'ont pas fourni de résultats expérimentaux.

On note finalement que cette extension, de même que l'algorithme *Reduce Candidate*, excluent les intervalles communs de taille 1 ce qui empêche un traitement général. On verra au prochain chapitre que la levée de cette restriction permet la construction d'algorithmes beaucoup plus simples.

CHAPITRE II

GÉNÉRATEURS

Nous présentons ici la notion des générateurs de Bergeron et al. (2005). Les générateurs et le vecteur *Support* (qui sera vu à la Section 3.3), sont deux notions sur lesquelles repose l'Algorithme 7 des mêmes auteurs qui permet de calculer, efficacement et simplement, les intervalles communs d'un ensemble de K permutations. Les générateurs permettent de plus une représentation en taille linéaire des $O(n^2)$ intervalles communs. Nous définissons d'abord les générateurs pour un ensemble de K permutations, puis nous décrivons un générateur particulier qui a l'avantage d'être facile à calculer. La méthode de calcul sera vue au chapitre III. Par la suite, nous montrons comment calculer aisément un générateur à partir d'autres générateurs et nous terminons avec une propriété importante de ce générateur, la commutation.

2.1 Générateurs pour les intervalles communs d'un ensemble de permutations

Les générateurs de Bergeron et al. (2005) sont des paires de vecteurs (R, L) de taille n où chaque valeur correspond à un intervalle. Cet intervalle, que l'on pourrait intuitivement nommer "intervalle commun à trous", est une version relâchée de l'intervalle commun. Dans le cas du vecteur R, l'intervalle commun à trous $(i..R[i])$ constitué des éléments situés entre i et $R[i]$ dans la permutation identité Id_n, peut inclure un ou des éléments plus grands que $R[i]$ dans une des autres permutations de l'ensemble $\mathcal{P}$. Ces éléments en trop sont en quelque sorte les trous de ce qui serait autrement un intervalle

commun. Similairement pour le vecteur L, l'intervalle commun à trous $(L[i]..i)$ constitué des éléments entre $L[i]$ et i dans Id_n, peut inclure un ou des éléments plus petits que $L[i]$ dans une autre permutation.

L'idée centrale est que si les valeurs $L[j]$, i, j et $R[i]$ sont en ordre croissant, $(i..j)$ est un intervalle commun si et seulement si l'intersection des intervalles $(i..R[i])$ et $(L[j]..j)$ donne $(i..j)$. Ainsi, plutôt que d'identifier directement les intervalles communs, on identifie les intervalles à trous de type L et R pour chacun des éléments de nos permutations, ce qui nous permettra par la suite d'engendrer aisément tous les intervalles communs. Nous verrons à la section 2.2 qu'il est facile de calculer un tel générateur mais d'abord, définissons plus formellement la notion de générateur.

Définition 3. Soit $\mathcal{P} = \{Id_n, P_2, \ldots, P_K\}$ un ensemble de K permutations sur n éléments. Un *générateur* pour les intervalles communs de $\mathcal{P}$ est une paire (R, L) de vecteurs de taille n tel que :

1. $R[i] \geq i$ et $L[j] \leq j, \forall i \forall j \in \{1, 2, \ldots, n\}$,
2. $(i..j)$ est un intervalle commun de $\mathcal{P}$ si et seulement si $(i..j) = (i..R[i]) \cap (L[j]..j)$.

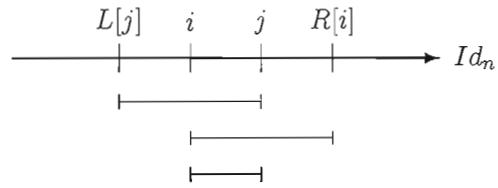


FIG. 2.1 Les intervalles $(L[j]..j)$ et $(i..R[i])$ sur Id_n d'un générateur (R, L). En bas et en gras, le résultat de $(i..R[i]) \cap (L[j]..j)$. Puisque $L[j] \leq i \leq j \leq R[i]$, l'intersection donne $(i..j)$.

La Figure 2.1 montre une représentation graphique de $(i..j) = (i..R[i]) \cap (L[j]..j)$. Par exemple, soit $\mathcal{P} = \{(1, 2, 3, 4), (3, 4, 1, 2)\}$. Nous pouvons aisément distinguer à l'œil que les intervalles communs non triviaux sont (1..2) et (3..4), ce qui donne un total de sept intervalles communs, si nous incluons les cinq intervalles triviaux. Un générateur valide serait $R = (4, 3, 4, 4)$ et $L = (1, 1, 3, 1)$ car

1. $R[i] \geq i$ et $L[j] \leq j$, $\forall i \forall j \in \{1, 2, 3, 4\}$ et

2. Le tableau ci-dessous vérifie que $(i..j)$ est un intervalle commun de $\mathcal{P}$ si et seulement si $(i..j) = (i..R[i]) \cap (L[j]..j)$.

Intervalle $(i..j)$	$(i..R[i]) \cap (L[j]..j)$	Intervalle commun	Remarque
(1..1)	$(1..4) \cap (1..1) = (1..1)$	Oui	trivial
(1..2)	$(1..4) \cap (1..2) = (1..2)$	Oui	non trivial
(1..3)	$(1..4) \cap (3..3) = (3..3)$	Non	$(1..3) \neq (3..3)$
(1..4)	$(1..4) \cap (1..4) = (1..4)$	Oui	trivial
(2..2)	$(2..3) \cap (1..2) = (2..2)$	Oui	trivial
(2..3)	$(2..3) \cap (3..3) = (3..3)$	Non	$(2..3) \neq (3..3)$
(2..4)	$(2..3) \cap (1..4) = (2..3)$	Non	$(2..4) \neq (2..3)$
(3..3)	$(3..4) \cap (3..3) = (3..3)$	Oui	trivial
(3..4)	$(3..4) \cap (1..4) = (3..4)$	Oui	non trivial
(4..4)	$(4..4) \cap (1..4) = (4..4)$	Oui	trivial

La première colonne contient toutes les valeurs possibles de $(i..j)$ sur Id_n où $i \leq j$, la deuxième colonne détaille le calcul de $(i..R[i]) \cap (L[j]..j)$ pour l'intervalle $(i..j)$ de la ligne courante, la troisième colonne indique si $(i..j)$ est un intervalle commun. C'est le cas chaque fois que $(i..R[i]) \cap (L[j]..j) = (i..j)$. Dans le cas où un intervalle commun existe, la dernière colonne informe s'il s'agit d'un intervalle trivial ou non et dans l'autre cas, on indique l'inégalité justifiant l'absence d'intervalle commun.

À noter que les trois générateurs suivants donnent exactement les même résultats : (R, L), (R', L) et (R', L') où $R' = (4, 2, 4, 4)$ et $L' = (1, 1, 2, 1)$.

Remarque 2. Les générateurs sont loin d'être uniques. Par exemple pour l'ensemble $\mathcal{P}$ de l'exemple ci-dessus, trois différents générateurs sont valides malgré que la taille n des permutations n'est que de 4.

La Figure 2.2 donne une représentation graphique d'un générateur. À juste titre,

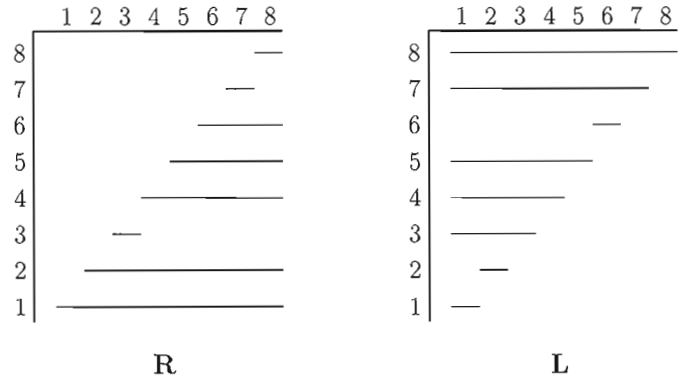


FIG. 2.2 Représentation graphique d'un générateur (R, L) pour l'ensemble $\{Id_n, P_2\}$ où $P_2 =$ (1 3 2 4 5 7 6 8), $R =$ (8, 8, 3, 8, 8, 8, 7, 8) et $L =$ (1, 2, 1, 1, 1, 6, 1, 1). Dans la figure de gauche, une ligne i s'étend de la colonne i à la colonne $R[i]$ et dans la figure de droite, une ligne i s'étend de la colonne $L[i]$ à la colonne i.

on pourrait se demander si un générateur existe pour tout ensemble de permutations. Nous verrons à la Proposition 2 de la section suivante que c'est bien le cas.

Puisqu'un générateur n'est composé que deux vecteurs de taille n, l'espace qu'il occupe est linéaire. Nous verrons à la Section 3.3 que les $O(n^2)$ intervalles communs s'obtiennent par un algorithme très simple, dans un temps proportionnel au nombre d'intervalles communs. Ceci rend alors inutile la conservation de l'ensemble de tous les intervalles communs identifiés pour un ensemble de permutations. Il suffit de conserver le générateur et d'engendrer les intervalles communs au besoin. Ainsi, tel que discuté en introduction, dans le cas d'une comparaison des génomes très semblables comme ceux de l'Homme et du chimpanzé, on pourrait représenter 1.6 milliards d'intervalles communs par simplement deux vecteurs de taille 40 000 chacun.

2.2 Un générateur facile à calculer

Bien que plusieurs générateurs existent pour les intervalles communs d'un ensemble $\mathcal{P}$ de K permutations (voir Section 2.1), certains se calculent plus facilement avec des algorithmes efficaces. C'est le cas du générateur (Sup, Inf) présenté dans cette section.

Définition 4. Soit $P = (p_1, p_2, \ldots, p_n)$ une permutation sur n éléments. Pour chaque élément p_i, on définit deux intervalles qui contiennent p_i :

$IMax[p_i]$ est le plus grand intervalle de P où chacun de ses éléments est $\geq p_i$,

$IMin[p_i]$ est le plus grand intervalle de P où chacun de ses éléments est $\leq p_i$.

On définit aussi :

$Sup[p_i]$ est le plus grand entier tel que $(p_i..Sup[p_i]) \subseteq IMax[p_i]$,

$Inf[p_i]$ est le plus petit entier tel que $(Inf[p_i]..p_i) \subseteq IMin[p_i]$.

On se rappelle que la notation parenthésée indique que $(p_i..Sup[p_i])$ et $(Inf[p_i]..p_i)$ sont des intervalles sur la permutation identité. Ainsi, $(p_i..Sup[p_i]) \subseteq IMax[p_i]$ est une suite de nombres qui se succèdent dans Id_n, qui débute par p_i (par exemple 5, 6 et 7 pour $p_i = 5$) et qui est incluse dans l'intervalle $IMax[p_i]$ de la permutation P. De façon analogue, $(Inf[p_i]..p_i) \subseteq IMin[p_i]$ est une suite de nombres qui se succèdent dans Id_n, qui se termine par p_i (par exemple 3, 4 et 5 pour $p_i = 5$) et qui est incluse dans l'intervalle $IMin[p_i]$ de P.

Prenons comme exemple $P = (1, 4, 7, 5, 9, 6, 2, 3, 8)$. On a $Imax[5] = \{7, 5, 9, 6\}$ car cela constitue le plus grand intervalle de P qui contient 5 où tous les éléments sont supérieurs ou égal à 5. On a aussi $Sup[5] = 7$ car dans {7, 5, 9, 6}, la plus longue suite débutant par 5 qu'il est possible de trouver est {5, 6, 7} et son élément le plus grand est 7. De façon analogue, $IMin[8] = \{6, 2, 3, 8\}$ car cela constitue le plus grand intervalle de P qui comprend 8 où tous les éléments sont inférieurs ou égal à 8. Finalement, $Inf[8] = 8$ car dans {6, 2, 3, 8}, la plus longue suite se terminant par 8 qu'il est possible de trouver est {8} et son élément le plus grand est 8. Le tableau 2.1 présente toutes les valeurs de $IMax[p_i]$, $Sup[p_i]$, $IMin[p_i]$ et $Inf[p_i]$ pour cette permutation P.

Remarque 3. Pour une permutation quelconque sur l'ensemble $\{1, 2, \ldots, n\}$, on a nécessairement $Inf[n] = 1$ car n est le plus grand élément possible alors tous les autres éléments seront plus petits. On a aussi $Sup[1] = n$ car 1 est le plus petit élément possible alors tous les autres éléments seront plus grands.

Proposition 2. *Soit $\mathcal{P}$ un ensemble de deux permutations sur n éléments dont la per-*

TAB. 2.1 $IMax$, $IMin$, Sup et Inf pour $P = (1,4,7,5,9,6,2,3,8)$

p_i	$IMax[p_i]$	$Sup[p_i]$	$IMin[p_i]$	$Inf[p_i]$
1	1 4 7 5 9 6 2 3 8	9	1	1
2	4 7 5 9 6 2 3 8	9	2	2
3	3 8	3	2 3	2
4	4 7 5 9 6	7	1 4	4
5	7 5 9 6	7	5	5
6	9 6	6	6 2 3	6
7	7	7	1 4 7 5	7
8	8	8	6 2 3 8	8
9	9	9	1 4 7 5 9 6 2 3 8	1

mutation identité Id_n. La paire de vecteurs (Sup, Inf) est un générateur des intervalles communs de $\mathcal{P}$.

Démonstration. Par définition, la paire de vecteurs (Sup, Inf) est un générateur si les deux conditions suivantes sont respectées :

1. $Sup[i] \geq i$ et $Inf[j] \leq j$ pour tout $i, j \in \{1, 2, \ldots, n\}$ et
2. $(i..j)$ est un intervalle commun de $\mathcal{P}$ si et seulement si
   $(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$.

La première condition se vérifie directement par la définition de Sup et Inf. En effet, $Sup[i]$ est défini comme le plus grand entier tel que $(i..Sup[i]) \subseteq IMax[i]$ alors $Sup[i]$ est nécessairement $\geq i$. Similairement, $Inf[j]$ est le plus petit entier tel que $(Inf[j]..j) \subseteq IMin[j]$ alors $Inf[j]$ est nécessairement $\leq j$.

La deuxième condition (une biconditionnelle) se vérifie en deux étapes.

Montrer

1. que si $(i..j)$ est un intervalle commun de $\mathcal{P}$ alors

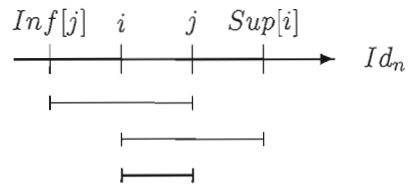


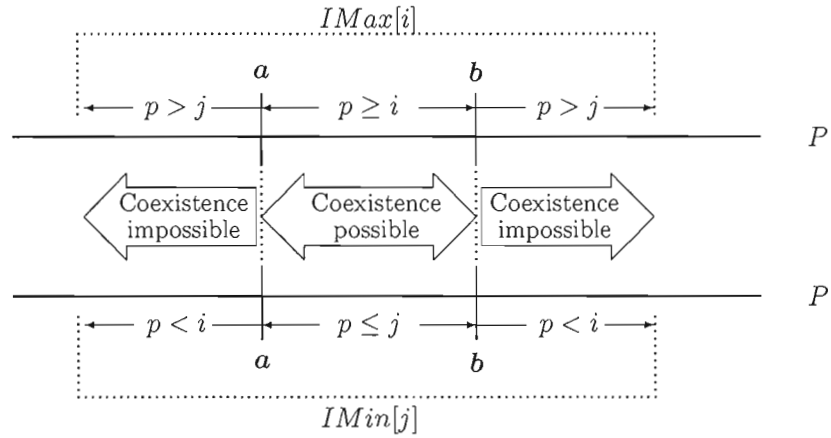
FIG. 2.3 Représentation sur Id_n des intervalles $(Inf[j]..j)$, $(i..Sup[i])$ et de leur intersection. Si $(i..j)$ est un intervalle commun de l'ensemble $\mathcal{P} = \{P, Id_n\}$, alors $i \geq Inf[j]$ et $j \leq Sup[i]$. Puisque $i \leq j$, on obtient $(i..Sup[i]) \cap (Inf[j]..j) = (i..j)$.

$(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$ et

2. que si $(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$ alors $(i..j)$ est un intervalle commun de $\mathcal{P}$.

Si $(i..j)$ est un intervalle commun de $\mathcal{P} = \{P, Id_n\}$, alors les éléments i à j forment un intervalle dans la permutation P. Autrement dit, il existe un intervalle de P qui contient tous les éléments p_i tel que $j \leq p_i \leq i$ et aucun autre. Ainsi $Sup[i] \geq j$ et $Inf[j] \leq i$. Puisque $i \leq j$ on a $Inf[j] \leq i \leq j \leq Sup[i]$ et du Lemme 1, on obtient $(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$ (voir Figure 2.3).

Si on a $(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$, alors du Lemme 1 on a $Inf[j] \leq i \leq j \leq Sup[i]$. Ainsi l'intervalle $IMax[i]$ contient tous les éléments de i à j puisque $Sup[i] \geq j$, tous les éléments de $IMax[i]$ sont par définition $\geq i$ et $IMax[i]$ contient possiblement des éléments $> j$. Par un raisonnement similaire, l'intervalle $IMin[j]$ contient aussi tous les éléments de i à j puisque $Inf[j] \leq i$, tous les éléments de $IMin[j]$ sont par définition plus petits ou égaux à j et $IMin[j]$ contient possiblement des éléments plus petits que i. On a donc deux intervalles qui coexistent sur la permutation quelconque P et qui contiennent chacun tous les éléments de i à j. Considérons un intervalle de P qui débute par le premier élément de l'ensemble $\{i, i+1, \ldots, j\}$ rencontré dans P et qui se termine par le dernier élément de ce même ensemble rencontré dans P. Puisque cet intervalle contient tous les éléments p tels que $i \leq p \leq j$, il est nécessairement inclus à la fois dans les intervalles $IMax[i]$et $IMin[j]$ et doit par conséquent satisfaire les contraintes de ces deux derniers, soit de n'avoir aucun élément $< i$ et aucun élément $> j$. Cet intervalle est donc forcément $(i..j)$ et il est commun (voir la Figure 2.4). □



a : position la plus à gauche où un élément de l'ensemble $\{i, i+1, \ldots, j\}$ est rencontré dans P.
b : position la plus à droite où un élément de l'ensemble $\{i, i+1, \ldots, j\}$ est rencontré dans P.

FIG. 2.4 L'intersection de $IMax[i]$ et $IMin[j]$ dans la permutation P donne obligatoirement l'intervalle $(i..j)$ lorsque $Inf[j] \leq i \leq j \leq Sup[i]$. On suppose que tous les éléments p de P où $i \leq p \leq j$ sont situés inclusivement entre les positions a et b. Alors les éléments dans $IMax[i]$ dont la position est plus petite que a ou plus grande que b sont plus grands que j et pour ces mêmes positions, les éléments dans $IMin[j]$ sont plus petits que i. Pour ces positions, il ne peut donc y avoir intersection (coexistence) de $IMax[i]$ et $IMin[j]$ car il faudrait que les éléments soient à la fois plus grand que j et plus petit que i, ce qui est impossible sachant que $i \leq j$. L'intersection (coexistence) ne peut se réaliser qu'entre les positions a et b où il est possible d'avoir des éléments p où $p \geq i$ et $p \leq j$. Puisque les intervalles $IMax[i]$ et $IMin[j]$ couvrent sur P les positions entre a et b (voir texte), forcément, tous leurs éléments p doivent satisfaire la double contrainte $p \geq i$ et $p \leq j$. Comme tout élément $\in \{i, i+1, \ldots, j\}$ est entre a et b, on a $IMax[i] \cap IMin[j] = (i..j)$. Rappel : $(i..j)$ est un intervalle sur Id_n.

2.3 Générateurs pour plusieurs ensembles de permutations

Si on a plusieurs ensembles de permutations et un générateur des intervalles communs de chacun de ces ensembles, il est facile de calculer un générateur pour l'union de tous ces ensembles. Nous verrons d'abord comment on obtient un générateur à partir de 2 générateurs puis nous généraliserons pour le faire à partir de K générateurs.

2.3.1 Un générateur à partir de deux générateurs

Nous présentons d'abord un lemme qui servira à la démonstration de la Proposition 3 montrant comment on calcule un générateur à partir de deux générateurs.

Lemme 1. *On a* $(i..j) = (i..R[i]) \cap (L[j]..j)$ *si et seulement si* $L[j] \leq i \leq j \leq R[i]$.

Démonstration. Montrons d'abord que $(i..j) = (i..R[i]) \cap (L[j]..j) \Rightarrow L[j] \leq i \leq j \leq R[i]$.

$$\begin{aligned} (i..j) &= (i..R[i]) \cap ((L[j]..j) \\ &\Rightarrow (i..j) \subseteq (i..R[i]) \text{ et } (i..j) \subseteq (L[j]..j) \\ &\Rightarrow j \leq R[i] \text{ et } L[j] \leq i. \end{aligned}$$

Comme $i \leq j$, on obtient $L[j] \leq i \leq j \leq R[i]$. Nous montrons maintenant que $L[j] \leq i \leq j \leq R[i] \Rightarrow (i..j) = (i..R[i]) \cap (L[j]..j)$. Les quatre valeurs $L[j]$, i, j et $R[i]$ sont croissantes ce qui donne directement $(i..R[i]) \cap (L[j]..j) = (i..j)$. □

Soit deux vecteurs X et Y, on note $min(X, Y)$ le vecteur

$$(min(X[1], Y[1]), min(X[2], Y[2]), \ldots, min(X[n], Y[n]))$$

et $max(X, Y)$ le vecteur

$$(max(X[1], Y[1]), max(X[2], Y[2]), \ldots, max(X[n], Y[n])).$$

Proposition 3. *Soit* (R_1, L_1) *et* (R_2, L_2) *les générateurs des intervalles communs des deux ensembles* $\mathcal{P}_1$ *et* $\mathcal{P}_2$ *de permutations qui contiennent chacun la permutation identité*

Id_n. La paire $(min(R_1, R_2), max(L_1, L_2))$ est un générateur pour les intervalles communs de $\mathcal{P}_1 \cup \mathcal{P}_2$.

Démonstration. Notons (R, L) le générateur $(min(R_1, R_2), max(L_1, L_2))$. De la définition des générateurs (voir Définition 3), nous devons démontrer d'une part que $\forall i \forall j$, $R[i] \geq i$ et $L[j] \leq j$ et d'autre part, que $(i..j) = (i..R[i]) \cap (L[j]..j)$ si et seulement si $(i..j)$ est un intervalle commun de $\mathcal{P}_1 \cup \mathcal{P}_2$.

Puisque $R_1[i] \geq i$ et $R_2[i] \geq i$, on a $R[i] = min(R_1[i], R_2[i]) \geq i$. Similairement, $L_1[j] \leq j$ et $L_2[j] \leq j$ alors $L[j] = max(L_1[j], L_2[j]) \leq j$.

Le Lemme 1 montre que $(i..j) = (i..R[i]) \cap (L[j]..j) \Leftrightarrow L[j] \leq i \leq j \leq R[i]$. L'intervalle $(i..j)$ sera commun à $\mathcal{P}_1 \cup \mathcal{P}_2$ si et seulement si $(i..j)$ est un intervalle commun dans $\mathcal{P}_1$ et dans $\mathcal{P}_2$. Puisque (R_1, L_1) est un générateur des intervalles communs de $\mathcal{P}_1$, on a $(i..j) = (i..R_1[i]) \cap (L_1[j]..j)$ et du Lemme 1, $L_1[j] \leq i \leq j \leq R_1[i]$. De la même façon, (R_2, L_2) est un générateur des intervalles communs de $\mathcal{P}_2$ ce qui donne $(i..j) = (i..R_2[i]) \cap (L_2[j]..j)$ et du Lemme 1, $L_2[j] \leq i \leq j \leq R_2[i]$. Alors si $(i..j)$ est un intervalle commun de $\mathcal{P}_1 \cup \mathcal{P}_2$ on a

$$\begin{aligned}
(i..j) &= ((i..R_1[i]) \cap (L_1[j]..j)) \cap ((i..R_2[i]) \cap (L_2[j]..j)) \\
&= (i..R_1[i]) \cap (i..R_2[i]) \cap (L_1[j]..j) \cap (L_2[j]..j) \\
&= (i..min(R_1[i], R_2[i])) \cap (max(L_1[j], L_2[j])..j) \\
&= (i..R[i]) \cap (L[j]..j).
\end{aligned}$$

Inversement, si $(i..j) = (i..R[i]) \cap (L[j]..j)$ on a

$$\begin{aligned}
(i..j) &= (i..R[i]) \cap (L[j]..j) \\
&= (i..min(R_1[i], R_2[i])) \cap (max(L_1[j], L_2[j])..j) \\
&= (i..R_1[i]) \cap (i..R_2[i]) \cap (L_1[j]..j) \cap (L_2[j]..j) \\
&= ((i..R_1[i]) \cap (L_1[j]..j)) \cap ((i..R_2[i]) \cap (L_2[j]..j)),
\end{aligned}$$

alors $(i..j)$ est un intervalle commun de $\mathcal{P}_1$ et de $\mathcal{P}_2$, i.e. de $\mathcal{P}_1 \cup \mathcal{P}_2$. □

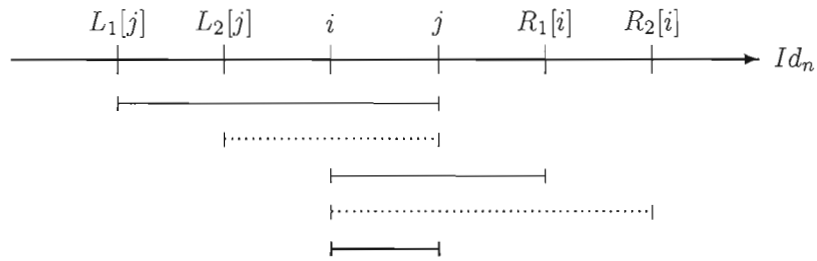


FIG. 2.5 Représentation sur Id_n de $(i..j) = ((L_1[j]..j) \cap (L_2[j]..j)) \cap ((i..R_1[i]) \cap (i..R_2[i]))$ où (R_1, L_1) et (R_2, L_2) sont les générateurs des intervalles communs des deux ensembles de permutations qui contiennent chacun Id_n. Graphiquement, il est aisé de constater que l'intervalle $(i..j)$ commun à ces deux ensembles ne peut s'obtenir qu'en prenant le plus grand entre $L_1[j]$ et $L_2[j]$ et le plus petit entre $R_1[i]$ et $R_2[i]$.

2.3.2 Un générateur à partir de K générateurs

Nous verrons dans le chapitre III suivant qu'il est possible de calculer un générateur pour les intervalles communs de deux permutations en temps $O(n)$. Si on a K ensembles de deux permutations, chacun contenant la permutation identité Id_n, on peut calculer un générateur pour chacun de ces ensembles en temps $O(n)$ et au total, le temps de calcul des K générateurs est dans $O(Kn)$. De la Proposition 1, nous pouvons en déduire que le temps pour calculer un générateur des intervalles communs de toutes ces permutations à partir des K générateurs sera $O(Kn)$, car il suffit de calculer la paire $(min(R_1, R_2, \ldots, R_K), max(L_1, L_2, \ldots, L_K))$, ce qui représente exactement $2n(K-1)$ comparaisons. Pour un i fixé, on a $(min(R_1[i], R_2[i], \ldots, R_K[i])$ ce qui nécessite $(K-1)$ comparaisons et $n(K-1)$ pour les n valeurs de i. On répète ce calcul de façon similaire pour $max(L_1, L_2, \ldots, L_K))$ pour un total de $2n(K-1)$ comparaisons.

2.4 Générateurs commutants

Nous définissons ici la propriété de commutation, puis nous démontrons que les générateurs (Sup, Inf) sont commutants. Pour terminer, nous montrons que la construction d'un générateur à partir de K générateurs commutants donne aussi un générateur

commutant.

Définition 5. Deux ensembles A et B sont *commutants* si $A \subseteq B$ ou $B \subseteq A$ ou $A \cap B = \varnothing$. Une collection $\mathcal{C}$ d'ensembles est commutante si, pour toute paire d'ensembles A et B de $\mathcal{C}$, A et B sont commutants. Un générateur (R, L) pour les intervalles communs de $\mathcal{P} = \{Id_n, P_2, \ldots, P_K\}$ est commutant si les deux collections $\{(i..R[i])\}_{i \in (1..n)}$ et $\{(L[i]..i)\}_{i \in (1..n)}$ sont commutantes.

Le Lemme 2 ci-dessous servira à la Proposition 4, vue ensuite, qui démontre que le générateur (Sup, Inf) est commutant.

Lemme 2. *Si $j \in IMin[k]$ alors $IMin[j] \subseteq IMin[k]$ et similairement, si $j \in IMax[k]$ alors $IMax[j] \subseteq IMax[k]$.*

Démonstration. Examinons la Figure 2.6 : on a l'intervalle $IMin[k]$ formé d'un ensemble d'éléments $p \leq k$. Une façon intuitive d'obtenir $IMin[k]$ est de partir de l'élément k, de parcourir P de chaque côté de l'intervalle $\{k\}$ et d'étendre cet intervalle jusqu'à ce qu'un élément $p > k$ soit rencontré. Si $j \in IMin[k]$, on a alors $IMin[j] \subseteq IMin[k]$. En effet, $IMin[j]$ ne peut s'étendre au delà de $IMin[k]$ car les éléments p_a et p_d qui sont plus grands que k et qui empêchent $IMin[k]$ de s'étendre d'avantage, empêchent aussi l'expansion de $IMin[j]$ ($IMin[j]$ ne peut inclure des éléments $> j$). Au mieux, l'intervalle $IMin[j]$ comprendra tous les éléments de $IMin[k]$ et au pire, $IMin[j]$ ne comprendra que l'élément j. Ainsi $IMin[j]$ est un sous-ensemble de $IMin[k]$ lorsque $j \in IMin[k]$. La preuve pour $IMax$ est similaire. □

Proposition 4. *Le générateur (Sup, Inf) des intervalles communs des permutations Id_n et P est commutant.*

Démonstration. Soit les deux intervalles $(Inf[k]..k)$ et $(Inf[j]..j)$ où, sans perte de généralité, on suppose $j < k$. Pour prouver que ces intervalles sont commutants, on

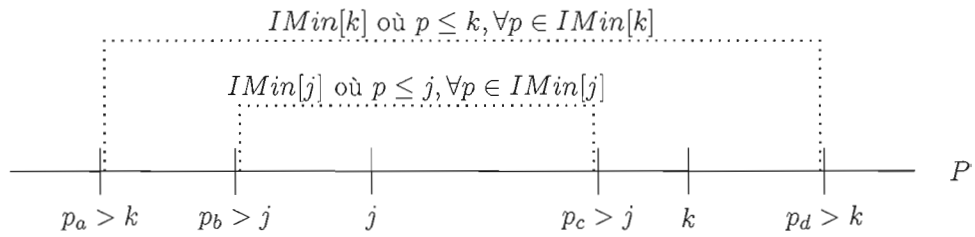


FIG. 2.6 Représentation des intervalles $IMin[k]$ et $IMin[j]$ sur la permutation P où $j \in IMin[k]$. Le premier élément de l'intervalle $IMin[k]$ est situé immédiatement à la droite de l'élément p_a et le dernier, immédiatement à la gauche de l'élément p_d. De la même façon, le premier élément de l'intervalle $IMin[j]$ est situé immédiatement à la droite de p_b et le dernier, immédiatement à la gauche de p_c. $IMin[j]$ est un sous-ensemble propre de $IMin[k]$, car les éléments p_a et p_d qui sont plus grands que k, donc plus grand que j aussi, empêchent $IMin[j]$ de s'étendre au delà des limites de $IMin[k]$ et par hypothèse, $k \notin IMin[j]$. L'expansion de $IMin[j]$ est empêchée par tout élément $p > j$ ce qui est au moins le cas pour k, p_a et p_d. Alors forcément, $IMin[j] \subseteq IMin[k]$.

doit montrer que si $(Inf[k]..k) \cap (Inf[j]..j) \neq \varnothing$ alors $(Inf[j]..j) \subset (Inf[k]..k)$. On a que $j \in (Inf[k]..k)$ puisque l'intersection est non vide et $j < k$. On va démontrer que $(Inf[k]..k)$ contient aussi $Inf[j]$ impliquant que $(Inf[j]..j)$ est inclus dans $(Inf[k]..k)$. Comme $j \in (Inf[k]..k)$, on a $j \in IMin[k]$ et du Lemme 2, on obtient $IMin[j] \subseteq IMin[k]$. Alors $(Inf[j]..j) \in IMin[k]$ et puisque par hypothèse $j < k$, on a $(Inf[j]..j) \subset (Inf[j]..k)$. Puisque $Inf[k]$ est minimal dans $(Inf[k]..k)$, on doit avoir $Inf[k] \leq Inf[j]$ et ainsi $(Inf[j]..j) \subset (Inf[k]..k)$. La preuve est similaire pour Sup. □

2.4.1 Générateurs commutants à partir d'autres générateurs commutants

Le Lemme 3 présenté ci-dessous servira à la Proposition 5 qui suit. Cette proposition montre qu'un générateur construit à partir de deux générateurs commutants est aussi commutant.

Lemme 3. *Si $a \leq b$ et $a' \leq b'$ alors $min(a, a') \leq min(b, b')$.*

Démonstration. Deux cas sont possibles : soit $min(a, a') = a$, soit $min(a, a') = a'$.

Si $min(a, a') = a'$ alors $a' \leq a$. Par hypothèse on a $a \leq b$ et puisque $a' \leq a$ on obtient $a' \leq a \leq b$ soit $a' \leq b$. Comme par hypothèse on a $a' \leq b'$, on obtient $a' \leq min(b, b')$.

Si $min(a, a') = a$ alors $a \leq a'$. Par hypothèse on a $a' \leq b'$ et puisque $a \leq a'$ on obtient $a \leq a' \leq b'$ soit $a \leq b'$. Comme par hypothèse on a $a \leq b$, on obtient $a \leq min(b, b')$.

De ces deux cas possibles on obtient $min(a, a') \leq min(b, b')$. □

Proposition 5. *Soit (R_1, L_1) un générateur commutant et (R_2, L_2) un autre générateur commutant. Le générateur $(min(R_1, R_2), max(L_1, L_2))$ construit à partir de ces deux générateurs sera aussi commutant.*

Démonstration. Soit $(i..min(R_1[i], R_2[i]))$ et $(j..min(R_1[j], R_2[j]))$, $i < j$, une paire quelconque d'intervalles déterminés par le vecteur $min(R_1, R_2)$. Puisque R_1 commute, on a

1. $R_1[i] < j$ ou
2. $R_1[i] \geq R_1[j]$,

et de la même façon pour R_2 on a

3. $R_2[i] < j$ ou
4. $R_2[i] \geq R_2[j]$.

Ceci donne lieu à quatre possibilités lorsque l'on considère R_1 et R_2 (1 et 3, 1 et 4, 2 et 3 ou 2 et 4). Démontrer que les intervalles déterminés par le vecteur $min(R_1, R_2)$ commutent revient à démontrer que pour chacune de ces possibilités, on obtient $min(R_1[i], R_2[i]) < j$ ou $min(R_1[i], R_2[i]) \geq min(R_1[j], R_2[j])$. Examinons chacune de ces possibilités :

1 et 3 : $R_1[i] < j \wedge R_2[i] < j \Rightarrow min(R_1[i], R_2[i]) < j$.

1 et 4 : On sait par définition que $min(R_1[j], R_2[j]) \geq j$ et par hypothèse que $R_2[i] \geq R_2[j]$ ce qui donne $R_2[i] \geq R_2[j] \geq j$ soit $R_2[i] \geq j$. Par hypothèse, $R_1[i] < j$

et du résultat $R_2[i] \geq j$ on obtient $R_1[i] < j \leq R_2[i]$, soit $R_2[i] > R_1[i]$ et ainsi $min(R_1[i], R_2[i]) < j$.

2 et 3 : On sait par définition que $min(R_1[j], R_2[j]) \geq j$ et par hypothèse que $R_1[i] \geq R_1[j]$ ce qui donne $j \leq R_1[j] \leq R_1[i]$ soit $R_1[i] \geq j$. Par hypothèse, $R_2[i] < j$ et du résultat $R_1[i] \geq j$ on obtient $R_2[i] < j \leq R_1[i]$, soit $R_2[i] < R_1[i]$ et ainsi $min(R_1[i], R_2[i]) < j$.

2 et 4 : Par hypothèse on a $R_1[j] \leq R_1[i]$ et $R_2[j] \leq R_2[i]$. Du Lemme 3, on obtient directement $min(R_1[i], R_2[i]) \geq min(R_1[j], R_2[j])$.

□

On peut généraliser ce résultat pour la construction d'un générateur à partir de K générateurs commutants. Le générateur ainsi obtenu sera évidemment commutant.

CHAPITRE III

CALCUL DU GÉNÉRATEUR (SUP, INF)

Nous expliquons dans ce chapitre comment calculer le générateur (Sup, Inf) selon Bergeron et al. (2005) puis nous décrivons ensuite une nouvelle méthode qui généralise ce calcul.

3.1 Un algorithme efficace

Calculer le générateur (Sup, Inf) revient à trouver pour chacun des n intervalles $IMax[k]$, la valeur $Sup[k]$ et pour chacun des n intervalles $IMin[k]$, la valeur $Inf[k]$. On doit donc d'abord calculer les vecteurs $IMax$ et $IMin$ ce qui va nous permettre de calculer les vecteurs Sup et Inf constituant le générateur recherché. Afin d'alléger le propos, nous supposerons que l'on peut trouver si un élément fait partie d'un intervalle de P en un temps constant. Nous verrons comment y arriver par une simple méthode décrite à la Section 3.1.3.

Rappelons que pour une permutation P, $IMin[k]$ est le plus grand intervalle de P où chacun de ses éléments est $\leq k$. À partir de l'élément k, on pourrait naïvement parcourir P vers la gauche et vers la droite, tant que l'élément courant est plus petit que k, afin de déterminer les bornes de $IMin[k]$. En réitérant pour les n valeurs possibles de k, on obtient un algorithme quadratique selon n. Heureusement, il y a moyen de faire beaucoup mieux.

3.1.1 Calcul de $IMin$ et $IMax$

Bergeron et al. (2005) ont proposé un algorithme linéaire efficace pour calculer les bornes de chaque intervalle des vecteurs $IMin$ et $IMax$. Ils ont défini les vecteurs $LMin$ et $RMin$ qui contiennent respectivement les bornes gauches et droites des intervalles de $IMin$ et similairement, les vecteurs $LMax$ et $RMax$ pour les bornes des intervalles de $IMax$. Une variante différente de cet algorithme (voir Figure 3.1) effectue le calcul pour chacun de ces quatre vecteurs. Nous allons examiner en détails le calcul de $LMax$, sans toutefois décrire le calcul des autres bornes qui est très similaire.

Algorithme 1 Calcul du vecteur $LMax$

S est une pile vide et s, le dessus de cette pile.
S.empiler(0), $P[0] \leftarrow 0$
Pour i de 1 à n **Faire**
  Tant que $P[s] > P[i]$ **Faire**
    S.dépiler()
  Fin Tant que
  $LMax[P[i]] \leftarrow s + 1$
  S.empiler(i)
Fin Pour

Définition 6. Étant donné une position i dans une permutation P, on définit la position i^*, telle que p_{i*} est l'élément plus petit que p_i qui est le plus près, à gauche de p_i. Si un tel élément n'existe pas alors $i^* = 0$.

Au départ, on a une pile S de positions qui ne contient que la valeur 0 et on définit $p_0 = 0$. L'algorithme itère dans la permutation P sur les positions i de 1 à n qu'on accumule dans la pile (certaines seront possiblement éliminées) d'une telle façon qu'au début de chaque itération i, S contient invariablement i^* (et possiblement d'autres positions $\neq i^*$). Au début de chaque itération, les positions sont dépilées tant que i^* n'est pas atteint. Une fois i^* au dessus de la pile, on calcule $LMax[p[i]]$ qui vaut simplement

$i^* + 1$, puis on empile la position i courante.

Proposition 6. *Étant donné un ensemble de permutations $\mathcal{P} = \{Id_n, P\}$, l'algorithme 1 calcule $LMax$ de $\mathcal{P}$ dans un temps $O(n)$.*

Démonstration. La preuve repose sur la préservation de l'invariant de boucle suivant : S contient i^* au début de chaque itération i. Dans une premier temps, supposons que la pile S, initialisée à 0, ne soit jamais dépilée. Si on inclut $p_0 = 0$, par définition, i^* pour une position i donnée ($1 \leq i \leq n$) est inférieur à i. Puisque la fin de chaque itération empile la position i courante, S contient toutes les positions $j < i$ au début de chaque itération i. S contient donc i^* au début de chaque itération.

Considérons maintenant le fait que certaines positions soient dépilées. Posons s le dessus de la pile S. On dépile au début de chaque itération i toutes les positions s (nécessairement $< i$) où $p_s > p_i$. Si $p_s > p_i$, s ne peut être, par définition, une position i^* pour p_i. Il ne peut être non plus une position i^* pour tout autre élément situé à la droite de p_i. Posons p_k un élément situé à la droite de p_i . On cherche la position i^* de p_k. Deux situations sont possibles. Soit $p_k < p_i$ ou $p_k > p_i$. (Rappel : Tous les éléments d'une permutation sont différents).

1. Si $p_k < p_i$, sachant que $p_s > p_i$, on en déduit que $p_k < p_s$ et alors $s \neq i^*$.
2. Si $p_k > p_i$:
   a) si $i = i^*$ alors $s \neq i^*$ car il ne peut y avoir qu'un seul i^*.
   b) Si $i \neq i^*$, par définition, il y a nécessairement un élément $p_{i*} < p_k$ situé entre p_i et p_k sinon il faudrait que $i = i^*$ (car $p_i < p_k$), contredisant ainsi l'hypothèse $i \neq i^*$.

Ainsi dans tous les cas où $p_i < p_s$, on a $s \neq i^*$ et par conséquent dépiler la position s sous la condition $p_i < p_s$ ne peut pas faire perdre une position i^* pour les éléments p_i suivants. Alors si i^* est dans la pile au début de l'itération i, il le sera encore après les dépilements. En empilant la position i courante à la fin de l'itération i, on assure pour les cas 1 et 2a, que i^* sera encore dans la pile pour le début de la prochaine itération.

Il ne reste plus qu'à initialiser la pile avec une position i^* valable pour le début de la première itération. La première itération s'effectue pour $i = 1$ et i^* correspond à l'indice situé immédiatement à gauche de la borne gauche de $IMax[p_1]$. Puisque cette borne gauche de $IMax[p_1]$ est nécessairement située à la position 1, 0 devient la seule valeur possible pour le i^* initial.

Complexité temporelle : Notons d'abord que jamais l'algorithme ne va réussir à dépiler le dernier élément s qui est la position 0 car $\forall i,\ 0 < p[i]$. Comme il y a en tout exactement $n + 1$ positions d'ajoutées à la pile vide, la boucle *tant que* ne peut dépiler au total plus de n fois. Le nombre total d'opérations est donc clairement linéaire par rapport à n. □

Les calculs de $Rmax$, $Lmin$ et $Rmin$ (voir Figure 3.1) ne sont que de simples variations de l'algorithme pour calculer $Lmax$, vu ci-dessus en détails. Dans chacun des quatre cas nous avons

- Initialisation d'une pile S et d'une valeur extrémale de P (soit $P[0]$ ou $P[n+1]$),
- Itérations sur les indices i de P (de 1 à n ou de n à 1) et à chaque itération,
  1. au début S est dépilée tant qu'une comparaison entre $P[s]$ et $P[i]$ est vraie,
  2. on calcule pour l'élément p_i courant une borne valant dessus de la pile ± 1,
  3. finalement on empile la position i courante.
- Préservation d'un invariant : Au début de chaque itération i, la pile S contient une position i^* (dont la définition est ajustée selon la borne calculée).

Pour obtenir un algorithme qui calculera $Lmax$, $Rmax$, $Lmin$ ou $Rmin$, il suffit de remplacer par des variables les divers éléments qui varient : la valeur initiale de la pile, la position extrémale de P et de sa valeur associée, la direction des itérations, l'opérateur de comparaison entre $P[s]$ et $P[i]$ et la valeur que l'on ajoute au dessus de la pile. Évidemment, le jeu de variables utilisé déterminera le vecteur obtenu.

Calcul de $LMax$.

S.empiler(0)
$P[0] \leftarrow 0$
Pour i de 1 à n
  Tant que $P[s] > P[i]$ Faire
    S.dépiler()
  Fin tant que
  $LMax[P[i]] \leftarrow s + 1$
  S.empiler(i)
Fin pour

Calcul de $RMax$.

S.empiler($n + 1$)
$P[n + 1] \leftarrow 0$
Pour i de n à 1
  Tant que $P[s] > P[i]$ Faire
    S.dépiler()
  Fin tant que
  $RMax[P[i]] \leftarrow s - 1$
  S.empiler(i)
Fin pour

Calcul de $LMin$.

S.empiler(0)
$P[0] \leftarrow n + 1$
Pour i de 1 à n
  Tant que $P[s] < P[i]$ Faire
    S.dépiler()
  Fin tant que
  $LMin[P[i]] \leftarrow s + 1$
  S.empiler(i)
Fin pour

Calcul de $RMin$.

S.empiler($n + 1$)
$P[n + 1] \leftarrow n + 1$
Pour i de n à 1
  Tant que $P[s] < P[i]$ Faire
    S.dépiler()
  Fin tant que
  $RMin[P[i]] \leftarrow s - 1$
  S.empiler(i)
Fin pour

FIG. 3.1 Variations sur l'algorithme 1 pour le calcul de $LMax$, $RMax$, $LMin$ et $RMin$, soit les bornes de $IMax$ et $IMin$.

3.1.2 Calcul de (Sup, Inf) à partir de $IMin$ et $IMax$

Algorithme 2 Calcul du générateur (Sup, Inf)

//Initialisations

$Inf[1] \leftarrow 1$, $Sup[n] \leftarrow n$

Pour k de 1 à n **Faire**

$m[k] \leftarrow k$

$M[k] \leftarrow k$

Fin Pour

//Calcul de Inf

Pour k de 2 à n **Faire**

Tant que $m[k] - 1 \in IMin[k]$ **Faire**

$m[k] \leftarrow m[m[k] - 1]$

Fin Tant que

$Inf[k] \leftarrow m[k]$

Fin Pour

//Calcul de Sup

Pour k de $n - 1$ à 1 **Faire**

Tant que $M[k] + 1 \in IMax[k]$ **Faire**

$M[k] \leftarrow M[M[k] + 1]$

Fin Tant que

$Sup[k] \leftarrow M[k]$

Fin Pour

En supposant les bornes de $IMin$ et $IMax$ connues, l'Algorithme 2 procède en trois phases pour calculer le générateur (Sup, Inf) : initialisations, calcul de Inf et calcul de Sup. Les vecteurs m et M servent respectivement à consigner les valeurs $Inf[k]$ et $Sup[k]$ calculées ou en cours de calcul. L'utilisation des vecteurs Inf et Sup est facultative et ne sert qu'à illustrer le moment où le calcul de $m[k]$ et $M[k]$ est complété car lorsque l'algorithme se termine, les vecteurs m et M sont respectivement identiques

aux vecteurs Inf et Sup. Les explications qui suivent ne concernent que le calcul de Inf car le calcul de Sup est très similaire.

L'algorithme 2 est basé sur l'observation suivante : si on a $(k'..k) \subseteq IMin[k]$ alors, par définition, $Inf[k] \leq k'$ et comme l'ensemble $\{(Inf[k]..k)\}$ des intervalles déterminés par le vecteur Inf est commutant, on a $Inf[k] \leq Inf[k']$ et $(Inf[k']..k) \subseteq IMin[k]$. L'idée est d'étendre le plus possible vers la gauche l'intervalle $(k'..k)$ sur Id_n où la plus petite valeur possible de $k' \in IMin[k]$ correspondra au $Inf[k]$ recherché. Ainsi, si $(m[k]..k) \subseteq IMin[k]$ et si $m[k]-1 \in IMin[k]$, on a $(m[k]-1..k) \subseteq IMin[k]$ ce qui implique que $Inf[k] \leq Inf[m[k]-1]$ et $(Inf[m[k]-1]..k) \subseteq IMin[k]$. En supposant, pour $k' < k$, que $Inf[k']$ soit connu, la valeur de $Inf[m[k]-1]$ est connue car $m[k]-1 < k$. On peut donc tester directement si $Inf[m[k]-1]-1 \in IMin[k]$ et si oui, étendre d'avantage l'intervalle jusqu'à $(Inf[Inf[m[k]-1]-1]..k)$. La répétition en boucle de ce test qui étend notre intervalle de départ conduit rapidement à la valeur de $Inf[k]$ recherchée.

Examinons de façon intuitive l'efficacité de l'Algorithme 2 par rapport à l'approche naïve brièvement décrite au début de la Section 3.1. Pour trouver la valeur $Inf[k]$, l'approche naïve recherche, tant qu'il y a succès, les éléments $k-1$, $k-2$, ..., 1 dans l'intervalle $IMin[k]$. En comparaison, l'algorithme 2 recherche aussi au départ si l'élément $k-1$ est présent dans $IMin[k]$ et si c'est le cas, plutôt que de rechercher l'élément $k-2$, il effectue un saut à $Inf[k-1]$ pour rechercher ensuite l'élément $Inf[k-1]-1$, ce qui fera éviter souvent plusieurs comparaisons inutiles. Cette économie permet l'obtention d'un algorithme linéaire plutôt que quadratique. Afin d'illustrer, prenons l'itération 8 de la Figure 3.2. Pour trouver $IMin[8]$, l'approche naïve testera successivement la présence dans $IMin[8]$ des éléments 7, 6, 5, 4, 3, 2 et 1. L'algorithme efficace va aussi tester la présence de l'élément 7, mais va ensuite sauter à $Inf[7] = 3$ et tester ensuite la présence de l'élément $Inf[7]-1 = 2$, évitant ainsi de comparer naïvement les valeurs 6, 5, 4 et 3.

Proposition 7. *Soit P une permutation sur n éléments. Si les bornes des intervalles $IMax[k]$ et $IMin[k]$ sont connues pour tout k, alors l'Algorithme 2 calcule le générateur*

(Sup, Inf) en temps $O(n)$.

Démonstration. L'algorithme suppose qu'au début de la k^e itération de la deuxième boucle *Pour*, on a $Inf[k'] = m[k']$ pour tout élément $k' < k$, i.e., que les valeurs de Inf sont connues et consignées dans m, pour chaque itération k' précédente. De plus, au début de chaque itération k, de la phase d'initialisation on a $m[k] = k$ et par définition, $k \in IMin[k]$. Au début de la première itération, $k = 2$, le seul $k' < k$ est 1, et on a bien $Inf[k'] = m[k']$ pour tout $k' < k$ car $Inf[1] = m[1] = 1$ (par définition $Inf[1] = 1$ et de l'initialisation, $m[1] = 1$). Par définition, $Inf[k] \leq k$ et avant d'entrer dans la boucle *Tant que*, on a $Inf[k] \leq m[k]$ puisque $m[k]$ a été initialisé à k. Si la condition d'entrée $m[k] - 1 \in IMin[k]$ de la boucle *Tant que* est vraie, alors $Inf[k] \leq m[k] - 1$ et $Inf[k] \leq Inf[m[k] - 1]$.

Examinons l'instruction d'affectation $m[k] \leftarrow m[m[k] - 1]$ de cette boucle. Nous savons de l'hypothèse de départ que les $Inf[k']$ des itérations k' précédentes sont connus et consignés dans m. Au premier tour de la boucle, $m[k] = k$ et $m[k] - 1 < k$ alors $m[m[k] - 1]$ est connu, vaut $Inf[m[k] - 1]$ et est, par définition, plus petit que $m[k] = k$. Si le test de la boucle donne vrai, on affecte alors la valeur $m[m[k] - 1]$ à $m[k]$, le prochain tour de boucle utilisera un $m[k]$ inférieur à celui du tour actuel, et une nouvelle affectation fera encore baisser la valeur du $m[k]$ car l'affectation vaut au maximum $m[k] - 1$. Ainsi, on affectera toujours à $m[k]$ des valeurs connues de Inf calculées lors des précédentes itérations sur k, de sorte que $m[m[k] - 1] = Inf[m[k] - 1]$.

L'affectation de $m[m[k] - 1]$ dans $m[k]$ conserve l'invariant $Inf[k] \leq m[k]$ puisque si $m[k] - 1 \in IMin[k]$, on a $Inf[k] \leq Inf[m[k] - 1]$. Si le test de la boucle donne faux, $m[k] - 1 \notin IMin[k]$ et forcément, $Inf[k] > m[k] - 1$. De l'invariant $Inf[k] \leq m[k]$ et de $Inf[k] > m[k] - 1$ on a $Inf[k] = m[k]$.

Ainsi, avant la première itération et à la fin de chaque itération k subséquente, on a $m[k] = Inf[k]$ ce qui nous permet d'obtenir $Inf[k]$ pour $\forall k$, soit le vecteur Inf. voir la trace de l'Algorithme 2 à la Figure 3.2. La démonstration du calcul de *Sup* est

similaire.

La complexité temporelle du calcul du vecteur Inf s'appuie sur le fait que la boucle *Tant que* est exécutée au total exactement $n-1$ fois. Considérons, à tout moment durant l'exécution de l'algorithme, l'ensemble I des intervalles $(m[k]..k)$ sur la permutation identité qui ne sont pas inclus dans d'autres intervalles du même type. Suite à la phase d'initialisations, on compte n de ces intervalles soit $I = \{(1..1), (2..2), \ldots, (n..n)\}$ et à la fin de l'algorithme, il ne reste dans I que l'intervalle $(m[n]..n) = (1..n)$. Chaque affectation de $m[k]$ fusionne deux intervalles consécutifs $(m[m[k]-1]..m[k]-1)$ et $(m[k]..k)$ de I, car le $m[k]$ courant prend la valeur $m[m[k]-1]$. Puisqu'au départ on a n intervalles, la boucle ne peut effectuer au total plus de $n-1$ de ces fusions. On peut visualiser ces dernières dans une trace de l'Algorithme 2 présentée à la Figure 3.2. La preuve de la complexité du calcul de Sup est similaire. □

3.1.3 Appartenance à un intervalle en temps constant

$IMin[k]$ est un intervalle sur la permutation P et peut donc être exprimé par deux indices $[i, j]$ de P représentant ses deux bornes i.e. les positions de début et de fin de cet intervalle. Afin d'évaluer en temps constant si un élément p_x de P est dans l'intervalle $IMin[k]$, il suffit de connaître la position x de p et de faire deux comparaisons avec les bornes i et j. Si $i \leq x \leq j$ alors $p_x \in IMin[k]$. L'usage de la permutation inverse, notée P^{-1}, permet d'obtenir directement la position de tout élément de P. Ainsi, si $i \leq P^{-1}[p_x] \leq j$ alors $p_x \in IMin[k]$. Évidemment, la même procédure s'applique pour les intervalles de $IMax$.

La permutation inverse P^{-1} se calcule par un simple parcours de la permutation P, en consignant la position occupée par chaque élément rencontré (voir Algorithme 3 et Figure 3.3). La complexité temporelle de ce prétraitement, qui n'a besoin d'être exécuté qu'une seule fois, est évidemment linéaire selon le nombre d'éléments de P.

k	Les intervalles $(m[k]..k)$ sur Id_n	Sous-ensemble I
1	(1..1)(2..2)(3..3)(4..4)(5..5)(6..6)(7..7)(8..8)	(1..1)(2..2)(3..3)(4..4)(5..5)(6..6)(7..7)(8..8)
2	(1..1)(2..2)(3..3)(4..4)(5..5)(6..6)(7..7)(8..8)	(1..1)(2..2)(3..3)(4..4)(5..5)(6..6)(7..7)(8..8)
3	(1..1)(2..2)(3..3)(4..4)(5..5)(6..6)(7..7)(8..8)	(1..1)(2..2)**(3..3)(4..4)**(5..5)(6..6)(7..7)(8..8)
4	(1..1)(2..2)(3..3)(3..4)(5..5)(6..6)(7..7)(8..8)	(1..1)(2..2)**(3..4)(5..5)**(6..6)(7..7)(8..8)
5	(1..1)(2..2)(3..3)(3..4)**(3..5)**(6..6)(7..7)(8..8)	(1..1)(2..2)**(3..5)(6..6)**(7..7)(8..8)
6	(1..1)(2..2)(3..3)(3..4)(3..5)**(3..6)**(7..7)(8..8)	(1..1)(2..2)**(3..6)(7..7)**(8..8)
7	(1..1)(2..2)(3..3)(3..4)(3..5)(3..6)**(3..7)**(8..8)	(1..1)(2..2)**(3..7)(8..8)**
8	(1..1)(2..2)(3..3)(3..4)(3..5)(3..6)(3..7)**(3..8)**	(1..1)**(2..2)(3..8)**
8	(1..1)(2..2)(3..3)(3..4)(3..5)(3..6)(3..7)**(2..8)**	**(1..1)(2..8)**
8	(1..1)(2..2)(3..3)(3..4)(3..5)(3..6)(3..7)**(1..8)**	**(1..8)**

FIG. 3.2 Trace de l'algorithme 2 pour le calcul de Inf de la permutation (2 8 3 4 5 6 1 7). La fin de chaque itération k est représentée sur une ligne qui affiche le contenu de l'ensemble de tous les intervalles $(m[k]..k)$ et celui de son sous-ensemble I qui ne contient que les intervalles $(m[k]..k)$ qui sont pas inclus dans d'autres intervalles $(m[k]..k)$. Dans la deuxième colonne, un intervalle en gras indique qu'une modification a été apportée par rapport à son état de la fin de l'itération k précédente. Les intervalles en gras de la troisième colonne indiquent ceux de l'ensemble I qui seront fusionnés au prochain k. La ligne $k = 1$ représente l'état initial des deux ensembles. Pour $k = 2$ et $k = 3$, la boucle *tant que* n'est pas exécutée, les valeurs de $m[k]$ initiales deviennent directement les valeurs de $Inf[k]$ et aucune fusion n'est effectuée dans l'ensemble I. À $k = 4$, on a $m[k] - 1 = m[4] - 1 = 3 \in IMin[4]$, alors on entre dans la boucle *tant que*, la valeur de $m[4]$ passe de 4 à $m[4 - 1] = 3$, l'intervalle $(m[k]..k) = (4..4)$ devient (3..4) ce qui entraîne la fusion des intervalles (3..3) et (4..4) dans I. La deuxième tentative d'entrer dans la boucle ne réussit pas car $m[4] - 1 = 3 - 1 = 2 \notin IMin[4]$, alors $m[4] = 3$ devient $Inf[4]$ et on passe au k suivant. Les itérations pour $k = 5$, 6 et 7 sont similaires à $k = 4$. À $k = 8$, on effectue trois tours de boucle *tant que* qui font passer $m[8] = 8$ à $m[8] = 3$, $m[8] = 3$ à $m[8] = 2$ et finalement $m[8] = 2$ à $m[8] = 1$. À chaque tour, il y a fusion de deux intervalles dans I qui ne compte plus, finalement, que l'intervalle (1..8).

Algorithme 3 Calcul de l'inverse de la permutation P

Pour i de 1 à n **Faire**

$P^{-1}[P[i]] \leftarrow i$

Fin Pour

	1	2	3	4	5	6	7	8
P	2	8	3	4	5	6	1	7
P^{-1}	7	1	3	4	5	6	8	2

FIG. 3.3 Exemple de l'inverse d'une permutation P (voir Algorithme 3)

3.2 Un nouvel algorithme plus général

Nous avons vu aux Sections 3.1.1 et 3.1.2 deux algorithmes efficaces où le premier calcule les vecteurs $IMin$ et $IMax$ à partir desquels le second calcule le générateur (Sup, Inf). Le nouvel algorithme dit de *projection*, décrit dans cette section, généralise les deux algorithmes précédents, tout en conservant la même simplicité et efficacité. Selon la nature des données fournies, la projection peut calculer aussi bien les vecteurs $IMin$ et $IMax$ que le générateur (Sup, Inf).

Définition 7. Étant donné un élément i d'une permutation P, la *projection* d'un intervalle I de P sur une permutation Q est le plus grand intervalle de Q qui contient i et dont tous les éléments font partie de I.

Lemme 4. *La projection des intervalles* $I(i) = (1..i)$ *de* $P = Id_n$ *correspond aux intervalles* $IMin[i]$ *de la permutation* Q.

Démonstration. La preuve découle directement de la définition de $IMin[i]$ (voir Définition 4), qui est le plus grand intervalle de Q qui contient i et dont tous les éléments font partie de $(1..i)$. □

Par exemple, avec $P = Id_9$, Q =(1 4 7 5 3 6 9 2 8) et $I(7) = (1..7)$, le plus

grand intervalle de Q qui contient 7 et dont tous les éléments font partie de (1..7) est $\{1,4,7,5,3,6\} = IMin[7]$.

Lemme 5. *La projection des intervalles $I(i) = IMin[i]$ d'une permutation P sur la permutation $Q = Id_n$ correspond aux intervalles $(Inf[i]..i)$ de la permutation Id_n.*

Démonstration. La preuve découle directement de la définition de $Inf[i]$ (voir Définition 4), où $(Inf[i]..i)$ est le plus grand intervalle de Id_n qui contient i, et dont tous les éléments font partie de $IMin[i]$. □

Prenons comme exemple les mêmes deux permutations de l'exemple précédent, sauf qu'ici, $P = (1\ 4\ 7\ 5\ 3\ 6\ 9\ 2\ 8)$ et $Q = Id_9$, et utilisons cette fois l'intervalle $I(7) = IMin[7] = \{1,4,7,5,3,6\}$. Le plus grand intervalle de Id_9 qui contient 7 et dont tous les éléments font partie de $IMin[7]$ est $(Inf[7]..7) = (3..7)$.

Ainsi, en effectuant deux projections successives, nous sommes en mesure d'obtenir, pour un élément i donné, la valeur $Inf[i]$. Il ne reste plus qu'à appliquer la projection sur un ensemble d'intervalles couvrant toutes les valeurs de i, afin de calculer les vecteurs $IMin$ et Inf. Nous verrons que les vecteurs $IMax$ et Sup s'obtiennent de façon similaire.

Définition 8. Soit P une permutation et $\{I(i)\} = \{I(1), \ldots, I(n)\}$ une famille d'intervalles de P. La famille $\{I(i)\}$ est *régulière* si

1. la famille $\{I(i)\}$ est commutante,
2. $\forall i, i \in I(i)$.

La *projection* de la famille $\{I(i)\}$ sur une permutation Q est notée $\Pi(\{I(i)\})$ et est la famille $\{J(i)\}$ telle que pour chaque i, $J(i)$ est la projection de l'intervalle $I(i)$.

L'Algorithme 4 calcule la projection d'un ensemble d'intervalles de la permutation P sur la permutation Q. La preuve de sa validité, démontrée à la Proposition 8 vue plus loin, s'appuie sur le Lemme 6 qui suit.

Algorithme 4 Projection Π de l'ensemble $\{I(i)\}$ de P sur Q

Précondition A) $\forall i$, i est le plus grand élément de $I(i)$ ou B) $\forall i$, i est le plus petit élément de $I(i)$

Précondition *Direction* ascendante si A), *Direction* descendante si B)

Précondition $\{I(i)\}$ est commutant et représenté dans un vecteur contenant les n intervalles $I(i)$ ordonnés de façon croissante sur i.

$\Pi(\{I(i)\}, P, Q, Direction)$ retourne $[(J_g(i), J_d(i))]$

P, Q : Deux permutations sur n éléments

$J_g(i)$, $J_d(i)$: Borne gauche et borne droite de l'intervalle $J(i)$ sur Q

// Initialisations

$Q^{-1} \leftarrow$ inverse(Q)

Pour i de 1 à n **Faire**

$g[i] \leftarrow Q^{-1}[i]$, $d[i] \leftarrow Q^{-1}[i]$

Fin Pour

Si *Direction* est ascendante **alors**

$Début \leftarrow 1$, $Fin \leftarrow n$

Sinon

$Début \leftarrow n$, $Fin \leftarrow 1$

Fin Si

// Calcul de $\{J(i)\}$

Pour i de $Début$ à Fin **Faire**

Tant que $Q[g[i]-1]]$ est dans $I(i)$ **Faire** //Étendre à gauche

$g[i] \leftarrow g[Q[g[i]-1]]$

Fin Tant que

$J_g[i] \leftarrow g[i]$

Tant que $Q[d[i]+1]]$ est dans $I(i)$ **Faire** //Étendre à droite

$d[i] \leftarrow d[Q[d[i]+1]]$

Fin Tant que

$J_d[i] \leftarrow d[i]$

Fin Pour

Lemme 6. *La projection d'une famille régulière est régulière.*

Démonstration. Par hypothèse, on sait que l'ensemble d'intervalles $\{I(i)\}$ commute et ainsi, pour tout i, j, on a un des trois cas suivants :

1. $I(i) \cap I(j) = \varnothing$ ou
2. $I(i) \subseteq I(j)$ ou
3. $I(j) \subseteq I(i)$.

On veut prouver que de ces trois cas, on obtient nécessairement que l'ensemble d'intervalles $\{J(i)\}$ commute i.e. que $\forall i \forall j$, $J(i) \cap J(j) = \varnothing$ ou $J(i) \subseteq J(j)$ ou $J(j) \subseteq I(i)$.

Par définition, $\Pi(I(i)) = J(i) \Rightarrow J(i) \subseteq I(i)$ et $\Pi(I(j)) = J(j) \Rightarrow J(j) \subseteq I(j)$. Alors dans le premier cas, on obtient $J(i) \cap J(j) = \varnothing$.

Dans le deuxième cas, deux situations sont possibles : Les intervalles $J(i)$ et $J(j)$ sont disjoints ou non. Si $J(i)$ et $J(j)$ sont disjoints, on a directement $J(i) \cap J(j) = \varnothing$. Si $J(i)$ et $J(j)$ ne sont pas disjoints alors $J(i) \cup J(j)$ est un intervalle dans la permutation Q puisque $J(i)$ et $J(j)$ sont des intervalles dans Q. De plus, cette union est incluse dans $I(j)$ car par définition, $J(i) \subseteq I(i)$ et par hypothèse, $I(i) \subseteq I(j)$ alors $J(i) \subseteq I(j)$ et puisque par définition $J(j) \subseteq I(j)$, on a $J(i) \cup J(j) \subseteq I(j)$. Par définition, $J(j)$ est le plus grand intervalle de la permutation Q inclus dans l'intervalle $I(j)$ de la permutation P. On obtient alors $J(i) \subseteq J(j)$. Le dernier cas est similaire au deuxième. □

Notons $J_g(i)$ la borne gauche de l'intervalle $J(i)$ et $J_d(i)$ sa borne droite.

Proposition 8. *L'algorithme 4 calcule la projection d'une famille d'intervalles réguliers dans un temps $O(n)$.*

Démonstration. La validité de l'algorithme repose sur la préservation d'un invariant de boucle. Prenons le cas de la direction ascendante et de la première boucle *tant que* qui effectue un parcours vers la gauche dans Q. L'algorithme se sert du vecteur g pour conserver les valeurs de J_g déjà calculées ou en cours de calcul. L'invariant est qu'au

début de chaque itération i, pour tout $i' < i$, $J_g[i'] = g[i']$. L'usage de la direction ascendante implique que les valeurs de i sont traitées de 1 à n. Ainsi, au début de toute itération i, pour $i' < i$, les valeurs $J_g[i'] = g[i']$ sont connues. Avant la première itération $i = 2$, le seul $i' < i$ est 1 et suite à la boucle d'initialisation, on a $g[1] = Q^{-1}[1]$ alors forcément, $J_g[1] = g[1]$ puisqu'il n'existe aucun élément plus petit que 1.

Au début de chaque itération i on a $g[i] = Q^{-1}[i]$ et $Q[g[i]] \in J(i)$. Si le test de la boucle *tant que* est vrai, c'est que l'élément $Q[g[i]-1]$, situé immédiatement à la gauche de l'élément $Q[g[i]]$, fait partie de $I(i)$. Puisque l'ensemble $\{I(i)\}$ commute et $i \in I(i)$, on a $I(Q[g[i]-1]) \subseteq I(i)$.

Ce résultat nous amène à deux cas possibles : soit $J(Q[g[i]-1]) \cap J(i) = \varnothing$, soit $J(Q[g[i]-1]) \subseteq J(i)$ (voir Lemme 6). Cependant, dans le présent contexte, le premier cas est impossible car l'élément $Q[g[i]-1]$ est situé immédiatement à gauche de l'élément $Q[g[i]] \in J(i)$, et puisque $Q[g[i]-1] \in I(i)$, on doit avoir $Q[g[i]-1] \in J(i)$, ce qui vient contredire l'hypothèse de l'intersection vide des intervalles $J(Q[g[i]-1)$ et $J(i)$. On a donc $I(Q[g[i]-1]) \subseteq I(i) \Rightarrow J(Q[g[i]-1]) \subseteq J(i)$, ce qui implique que $J_g(i) \leq J_g(Q[g[i]-1])$ et $g[i] \leq g[Q[g[i]-1]]$.

On affecte à la valeur de $J_g[i]$ en cours de calcul, conservée dans $g[i]$, la valeur $g[Q[g[i]-1]]$ et comme l'élément situé à ce nouveau $g[i]$ est $\in J(i)$, la boucle pourra refaire le même travail tant que le test d'entrée sera satisfait. Lorsque ce test échoue, l'élément situé immédiatement à gauche de la position $g[i]$ courante ne fait pas partie de $I(i)$. Puisqu'à ce moment $g[i]$ est la position la plus à gauche où $Q[g[i]] \in J(i)$, on a $J_g[i] = g[i]$ et l'invariant reste conservé. À la fin de la dernière itération, on a les valeurs de $J_g[i], \forall i$. La preuve est similaire pour le parcours ascendant vers la droite et pour la direction descendante.

L'analyse de complexité repose sur le nombre maximal de tours effectués par la boucle *tant que*. Prenons le cas de la direction ascendante avec parcours vers la gauche. Considérons l'ensemble des intervalles $\{(g[i]..Q^{-1}[i])\}$. Au départ, suite à l'étape d'initialisation, cet ensemble comprend les n intervalles $\{(1..1), (2..2), \ldots, (n..n)\}$. L'affectation

de la boucle *tant que* a pour effet de fusionner l'intervalle $(g[i]..Q^{-1}[i])$ courant avec le précédent, soit $(g[Q[g[i]-1]]..g[i]-1)$ (voir Figure 3.4). Puisqu'il ne peut y avoir plus de $n-1$ de ces fusions, au total, le nombre de tours de la boucle *tant que* est borné par $O(n)$. La preuve est similaire pour le parcours ascendant vers la droite et pour la direction descendante. □

À noter que le test d'appartenance d'un élément q de Q dans $I(i)$ se fait en temps constant en utilisant l'inverse de la permutation P, ce qui permet d'obtenir directement la position de tout élément de P. L'inverse de la permutation P, noté P^{-1}, est calculé en temps $O(n)$ par l'Algorithme 3 vu à la sous-section 3.1.3. En notant respectivement les bornes gauche et droite de chaque intervalle $I(i)$ de P par $I_g[i]$ et $I_d[i]$, on détermine que l'élément q fera partie de $I(i)$ si $I_g[i] \leq P^{-1}[q] \leq I_d[i]$.

Algorithme 5 Calcul du générateur (Inf, Sup) par projection

// Calcul de Inf

$I \leftarrow [(1..1),(1..2),\ldots,(1..n)]$

$IMin \leftarrow \Pi(I, Id_n, P, ascendante)$

$(Inf, Id_n) \leftarrow \Pi(IMin, P, Id_n, ascendante)$

// Calcul de Sup

$I \leftarrow [(1..n),(2..n),\ldots,(n..n)]$

$IMax \leftarrow \Pi(I, Id_n, P, descendante)$

$(Id_n, Sup) \leftarrow \Pi(IMax, P, Id_n, descendante)$

Proposition 9. *L'algorithme 5 calcule le générateur (Sup, Inf) pour un ensemble $\mathcal{P} = \{Id_n, P\}$ dans un temps $O(n)$.*

Démonstration. Pour calculer Inf, on fournit d'abord à l'algorithme de projection l'ensemble $\{I(i)\}$, dans un vecteur de la forme $[(1..i)] = [(1..1),(1..2),\ldots,(1..n)]$, ce qui satisfait toutes ses préconditions ($\{I(i)\}$ est commutant, i est le plus grand élément de $I(i)$, les intervalles $I(i)$ sont en ordre croissant de i et on utilise la direction ascendante).

i	Ensemble des intervalles $(g[i]..Q^{-1}[i])$	$g[i]$
1	(1..1),(2..2),(3..3),(4..4),(5..5),(6..6),(7..7),(8..8),(9..9)	1
2	(1..1),(2..2),(3..3),(4..4),(5..5),(6..6),(7..7),(8..8),(9..9)	8
3	**(1..1),(2..2)**,(3..3),(4..4),(5..5),(6..6),(7..7),(8..8),(9..9)	5
4	(1..2),(3..3),(4..4),(5..5),(6..6),(7..7),(8..8),(9..9)	~~2~~ 1
5	(1..2),(3..3),(4..4),**(5..5),(6..6)**,(7..7),(8..8),(9..9)	4
6	(1..2),(3..3),**(4..4),(5..6)**,(7..7),(8..8),(9..9)	~~6~~ 5
6	**(1..2),(3..3)**,(4..6),(7..7),(8..8),(9..9)	~~5~~ 4
7	(1..3),(4..6),(7..7),**(8..8),(9..9)**	~~3~~ 1
8	(1..3),**(4..6),(7..7)**,(8..9)	~~9~~ 8
9	**(1..3),(4..7)**,(8..9)	~~7~~ 4
9	(1..7),(8..9)	~~4~~ 1

FIG. 3.4 Trace de l'algorithme 1 pour la projection de l'ensemble commutant $\{I(i)\} = \{(1..i)\} = \{(1..1),(1..2),\ldots,(1..n)\}$ de la permutation Id_n sur $Q = (1\ 4\ 7\ 5\ 3\ 6\ 9\ 2\ 8)$. Puisque i est le plus grand élément dans chaque intervalle, la direction sera croissante. La fin de chaque itération i est représentée sur une ligne qui détaille le contenu correspondant de l'ensemble des intervalles $(g[i]..Q^{-1}[i])$. La dernière colonne montre les mises à jour sur $g[i]$ qui vaut initialement $Q^{-1}[i]$. Les intervalles en gras sont ceux qui seront fusionnés au prochain i. Pour $i = 2$, 3 et 5, la boucle *tant que* n'est pas exécutée, aucune fusion n'est effectuée et les valeurs de $g[i]$ initiales deviennent directement les valeurs de $J_g[i]$. À $i = 4$, l'élément situé immédiatement à la gauche de 4 dans Q est 1 et il fait partie de $I(i) = I(4) = (1..4)$. Alors on entre dans la boucle *tant que* et la valeur de $g[4]$ passe de 2 à $g[1] = 1$, l'intervalle $(g[i]..Q^{-1}[i]) = (2..2)$ devient (1..2) ce qui entraîne la fusion des intervalles (1..1) et (2..2). La deuxième tentative d'entrer dans la boucle ne réussit pas car il n'y a plus d'élément à la gauche de l'élément 1. Alors $g[4] = 1$ devient $J_g[4]$ et on passe au i suivant. Deux tours de boucle sont effectués pour $i = 6$ où il y a fusion successive de (5..5) et (6..6) puis de (4..4) et (5..6). La valeur finale de $g[6]$ est 4 puisque l'élément 7 situé à la position 3 ne fait pas partie de $I(6)$. Les autre itérations sont similaires.

Du Lemme 4, la projection retourne alors l'ensemble $\{J(i)\} = \{IMin[i]\}$. Au deuxième appel de projection, on utilise l'ensemble $\{I(i)\} = IMin$ qui satisfait aussi toutes les préconditions ($IMin$ est commutant, i est le plus grand élément de $IMin[i]$, les intervalles $IMin[i]$ sont en ordre croissant de i et on utilise la direction ascendante). Du Lemme 5, on obtient alors l'ensemble $\{J(i)\} = \{Inf[i]\}$. La preuve du calcul de Sup est similaire.

La complexité temporelle est directe puisque, de la Proposition 8, l'opération de projection s'effectue en temps linéaire. □

Nous pouvons appliquer ce dernier résultat à un ensemble $\mathcal{P}$ de K permutations, simplement en utilisant K fois l'Algorithme 5, soit une fois pour chacune des K permutations. On obtient alors K générateurs desquels on peut calculer un générateur représentant les intervalles communs de $\mathcal{P}$, tel que décrit à la section 2.3.2, en un temps $O(Kn)$. Ainsi, pour résumer, on a :

Théorème 1. *Étant donné $\mathcal{P} = \{Id_n, P_2, \ldots, P_K\}$ un ensemble de K permutations sur n éléments, un générateur pour les intervalles communs de $\mathcal{P}$ peut être calculé dans un temps $O(Kn)$.*

3.3 Génération des intervalles communs à partir d'un générateur

La précédente section a montré comment calculer un générateur pour un ensemble de K permutations en temps $O(Kn)$. Maintenant, à partir de ce générateur de taille $2n$, nous allons voir comment générer les $O(n^2)$ intervalles communs de cet ensemble. On crée d'abord en temps linéaire un vecteur intermédiaire nommé *Support*. Ce dernier permettra ensuite à un algorithme très simple d'énumérer les intervalles en un temps proportionnel au nombre d'intervalles communs.

Définition 9. Soit (R, L) un générateur commutant. On définit $Support[i]$, pour $i > 1$, comme le plus grand entier $j < i$ tel que $R[i] \leq R[j]$.

Une façon simple de visualiser un support est de se baser sur la représentation

graphique du vecteur R d'un générateur (R, L), tel qu'illustré dans l'exemple de la Figure 3.5. En s'imaginant une chute vers le bas des intervalles de la Figure 3.5a, on obtient la Figure 3.5b où un intervalle A est un support pour un intervalle B si A est directement sous B.

L'Algorithme 6 effectue le calcul du vecteur *Support* à partir d'un générateur qui doit être commutant. C'est le cas du générateur (Sup, Inf), tel que démontré à la Proposition 4.

Algorithme 6 Calcul du vecteur *Support*

Précondition Le générateur (R, L) commute

S est une pile vide
s est l'élément au dessus de la pile
S.empiler(1)
Pour i de 2 à n **Faire**
  Tant que $R[s] < i$ **Faire**
    S.dépiler
  Fin Tant que
  $Support[i] \leftarrow s$
  S.empiler(i)
Fin Pour

Proposition 10. *Étant donné un générateur commutant (R, L), l'Algorithme 6 effectue le calcul du vecteur Support dans un temps $O(n)$.*

Démonstration. La validité de l'algorithme repose sur l'invariant de la boucle *Pour* suivant : au début de toute itération i, la pile S contient $Support[i]$.

Cas de base : S est initialisée avec la valeur 1 et les itérations débutent avec $i = 2$. $Support[2]$ vaut nécessairement 1 et pour $i = 2$, le plus grand entier $j < i$ est unique et vaut 1. Ainsi pour $j = 1$, $R[j] = R[1] = n$ car tous les éléments sont plus grand que 1.

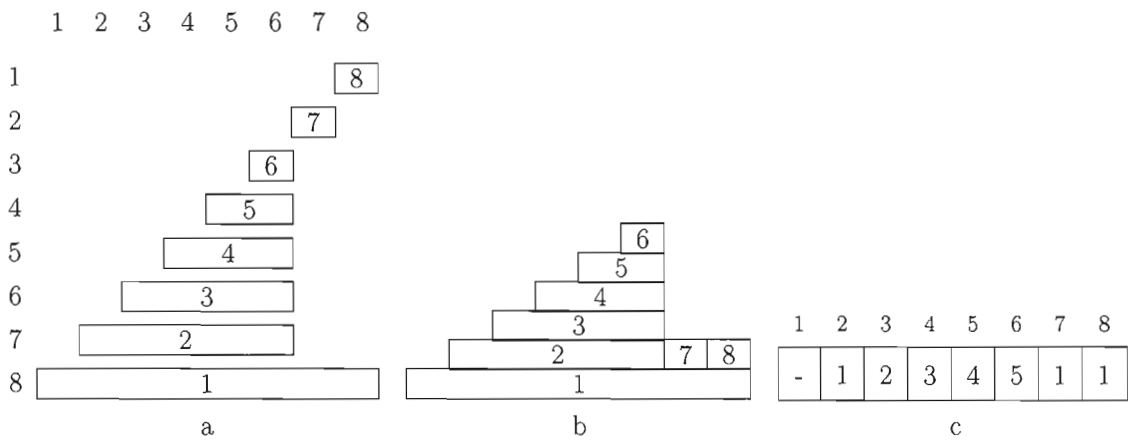


FIG. 3.5 a) Représentation graphique des intervalles déterminés par le vecteur $R = [8,6,6,6,6,6,7,8]$ d'un générateur (R, L) des intervalles communs de l'ensemble $\{Id_n, (2,8,3,4,5,6,1,7)\}$. b) Le résultat obtenu si on fait tomber vers le bas les intervalles de la figure a. Chaque intervalle qui est directement sous un autre est le support de ce dernier. Par exemple, l'intervalle 1 supporte les intervalles 2, 7 et 8 mais l'intervalle 3 ne supporte pas l'intervalle 5. c) Le vecteur support décrivant la situation de la figure b où $Support[i]$ est l'intervalle supportant l'intervalle i. Par exemple, $Support[2] = 1$ et $Support[5] = 4$.

Comme $R[1] = n$ est la plus grande valeur possible de $R[j]$ pour tout $1 \leq j \leq n$, $R[i]$ est nécessairement plus petit ou égal à $R[j]$. S initialisée avec 1 contient donc $Support[2]$ au début de la première itération.

Cas suivants : Dans une premier temps, supposons que la pile S ne soit jamais dépilée. Puisque la fin de chaque itération empile la position i courante, au début de l'itération i, S contient toutes les positions $j < i$. Par définition, $Support[i]$ est le plus grand entier $j < i$ qui respecte une certaine condition. Puisque S contient tous les $j < i$, S contient nécessairement $Support[i]$. Considérons maintenant le fait que certaines valeurs soient dépilées. Posons s le dessus de la pile S. On dépile au début de chaque itération i tous les s tel que $R[s] < i$, ce qui implique que $R[s] < R[i]$. Ainsi tout s dépilé ne peut être $Support[i]$. Puisque les itérations se font de 2 à n, les i suivants sont supérieur à i. Posons $k > i$ une itération subséquente à i. On a $R[s] < i < k$ et $R[s] < R[i] < R[k]$. Les s dépilés ne peuvent donc être support pour les i suivants et

l'invariant reste préservé.

Nous avons vu que l'invariant reste conservé i.e. qu'au début de toute itération i, la pile S contient $Support[i]$. Les itérations se font de 2 à n, S est initialisée avec 1 et le i courant est empilé dans S à la fin de chaque itération. Les valeurs dans S sont donc empilées en ordre croissant de i (de 1 à n) et les dépilements se font, de la nature même de la structure de pile, en ordre décroissant. Tel que vu dans l'explication de l'invariant, les s dépilés ne peuvent être $Support[i]$ pour le i courant et les suivants, et dès que la condition de dépilement est fausse, on a $R[s] < R[i]$. Ainsi s est le plus grand $j < i$ tel que $R[j] < R[i]$ i.e. $s = Support[i]$. Puisqu'à la fin de chaque itération i on obtient $Support[i]$, au terme de l'algorithme, le vecteur $Support$ sera entièrement calculé.

Complexité temporelle : jamais l'algorithme ne va dépiler son dernier élément qui est le 1 car $R[1] = n$ et $n \geq i$. Ainsi la boucle *Tant que*, où le dépilement est effectué, ne sera pas exécutée pour $s = 1$. Le nombre d'empilements est exactement de n, soit la valeur d'initialisation plus les $n-1$ empilements de la boucle *Pour*. Alors la boucle *Tant que* ne peut dépiler au total plus de $n - 1$ fois. Le nombre total d'opérations est donc dans $O(n)$.

□

Le Lemme 7 présenté ci-dessous servira à la démonstration du Théorème 2 présenté ensuite. Ce théorème prouve la validité de l'Algorithme 7 qui génère tous les intervalles communs d'un ensemble de K permutations.

Lemme 7. *Soit (R, L) un générateur des intervalles communs d'un ensemble $\mathcal{P}$ de permutations. Si $(i..j)$ est un intervalle commun de $\mathcal{P}$ alors $R[i] \geq R[j]$.*

Démonstration. $R[i]$ nous informe que les éléments i à $R[i]$ sont dans l'intervalle $Imax[i]$ et de la même façon, $R[j]$ nous informe que les éléments j à $R[j]$ sont dans l'intervalle $Imax[j]$. Puisque $(i..j)$ est un intervalle commun, les éléments i à j font partie de $Imax[i]$

Algorithme 7 Génération des intervalles communs

Pour j de 1 à n **Faire**

$i \leftarrow j$

Tant que $i \geq L[j]$ **Faire**

Produire l'intervalle $(i..j)$

$i \leftarrow Support[i]$

Fin Tant que

Fin Pour

et on a $R[i] \geq j$. Comme $j \geq i$ et $j \in Imax[i]$, $Imax[j] \subseteq Imax[i]$ (voir Lemme 2) et ainsi $R[i] \geq R[j]$. □

Théorème 2. *Étant donné un générateur commutant (R, L), l'Algorithme 7 génère tous les intervalles communs d'un ensemble $\mathcal{P}$ de K permutations sur n éléments dans un temps $O(n + N)$, où N est le nombre d'intervalles communs de $\mathcal{P}$.*

Démonstration. On suppose que l'intervalle $(i..j)$ est identifié par l'algorithme. Au début de la j^e itération de la boucle *pour*, $i = j$, ainsi $j \leq R[i]$. Si le test de la boucle *tant que* est vrai, alors $i \geq L[j]$ et $(i..j)$ est un intervalle commun puisque $L[j] \leq i \leq j \leq R[i]$ (voir Définition 3 et Lemme 1). Si $i' = Support[i]$, alors par définition $R[i'] \geq R[i]$, et on a donc $j \leq R[i']$ à la fin de chaque itération de la boucle *tant que*.

D'autre part, quand $i < j$, si $(i..j)$ est un intervalle commun de $\mathcal{P}$, nécessairement $Support[j] \geq i$ puisque du Lemme 7, $R[i] \geq R[j]$. Posons i' comme le plus petit entier tel que $i < i'$ et $(i'..j)$ un intervalle commun identifié par l'algorithme 7. Cet intervalle existe puisque $(j..j)$ est un intervalle commun. Comme R commute, $i < i' \leq j$ et $R[i] \geq R[j]$, on a $R[i] \geq R[i']$ et alors $Support[i'] \geq i$ (voir Figure 3.6). Mais supposons que $Support[i'] = i''$ pour $i < i'' < i'$. Dans ce cas, $R[i''] \geq R[i']$ et $R[i'] \geq j$ puisque par hypothèse, $(i'..j)$ est un intervalle commun. Alors $R[i''] \geq j$ et $(i''..j)$ est un intervalle commun ce qui contredit la définition de i'. On a donc $Support[i'] = i$.

Examinons la complexité temporelle. La boucle *pour* tourne obligatoirement n fois

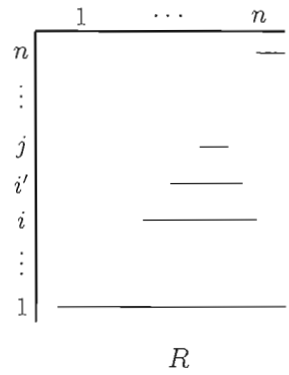


FIG. 3.6 Intervalles $(i..R[i])$ du générateur (R, L) où $(i..j)$ et $(i'..j)$ sont des intervalles communs. Si $i < i' \leq j$ et (R, L) est commutant, alors $R[i']$ ne peut être plus grand que $R[i]$. Sur la figure, l'intervalle $(j..R[j])$ est illustré à la ligne j, $(i'..R[i'])$ à la ligne i' et $(i..R[i])$ à la ligne i. On voit clairement que l'on doit avoir $R[i] \geq R[i']$ sinon $(i'..R[i'])$ chevaucherait $(i..R[i])$ ce qui rendrait R non commutant (voir démonstration du Théorème 2).

et au total, la boucle *tant que* tournera autant de fois qu'il y a des intervalles communs. Puisque le nombre possible d'intervalles communs est situé entre $n+1$ et n^2, la complexité de l'algorithme est au minimum $O(n)$ et au pire $O(N)$, soit $O(n + N)$. □

En fait, la complexité de cet algorithme est de $O(n^2)$ mais la formulation plus détaillée $O(n + N)$ a pour avantage de mettre en relief le fait que la complexité est linéaire par rapport aux entrées et sorties, donc optimale.

Cet algorithme est très efficace. Pour identifier les intervalles communs $(i..j)$, une approche naïve testerait l'existence des n^2 intervalles possibles en itérant sur chaque i et j. Pour un j donné, l'Algorithme 7 passe directement d'un intervalle commun à l'autre grâce à l'affectation $i \leftarrow \mathit{Support}[i]$, et limite à exactement n le nombre des tests qui ne mènent pas à un intervalle commun.

La grande simplicité, non seulement des algorithmes à utiliser (voir les Algorithmes 4, 5, 6 et 7), mais aussi des structures de données employées, soit quelques vecteurs et piles qui sont chacun de taille n, en font une solution optimalement efficace en pratique et extrêmement facile à implanter et à tester.

CHAPITRE IV

DES GÉNÉRATEURS CANONIQUES AUX INTERVALLES FORTS

Dans ce dernier chapitre, nous allons montrer comment calculer les intervalles communs forts à partir d'un type particulier de générateur que l'on nomme *générateur canonique*. Par la suite, nous ferons la description d'une représentation de ces intervalles à l'aide d'un arbre, ce qui permet de visualiser, en un seul coup d'oeil, les intervalles forts d'un ensemble de permutations. Nous terminerons par une discussion montrant les avantages de l'utilisation des intervalles forts, plutôt que des intervalles communs ordinaires, afin de détecter des groupes conservés entre les permutations.

4.1 Les générateurs canoniques

Nous avons vu, à la Remarque 2 de la Section 2.1, que les générateurs sont loin d'être uniques. Nous verrons qu'il est possible de définir un générateur de telle façon qu'il soit unique, toujours existant et commutant, et qu'il se calcule en temps linéaire à partir d'un autre générateur. Ce générateur particulier, en plus de permettre le calcul des intervalles communs, permet aussi de calculer les intervalles forts, tel que présenté à la Section 4.2.

Définition 10. Un générateur (R, L) des intervalles communs d'un ensemble $\mathcal{P}$ de K permutations est *canonique* si, $\forall i \in (1..n)$, les intervalles $(i..R[i])$ et $(L[i]..i)$ sont des intervalles communs.

Proposition 11. *Le générateur canonique des intervalles communs existe toujours, est unique et commutant.*

Démonstration. Pour $1 \leq i \leq n$, on définit $R[i]$ comme le plus grand entier tel que $(i..R[i])$ est un intervalle commun, et $L[i]$ comme le plus petit entier tel que $(L[i]..i)$ est un intervalle commun. Ces deux entiers existent puisque, par définition, $(i..i)$ est un intervalle commun. On a ainsi $R[i] \geq i$ et $L[i] \leq i$.

D'une part, si l'intervalle $(i..j)$ est un intervalle commun, alors $(i..j) \subseteq (i..R[i])$ et $(i..j) \subseteq (L[j]..j)$, ce qui implique que $L[j] \leq i \leq j \leq R[i]$, et ainsi (voir Lemme 1), $(i..j) = (i..R[i]) \cap (L[j]..j)$. D'autre part, si $(i..j) = (i..R[i]) \cap (L[j]..j)$, alors on a nécessairement $L[j] \leq i \leq j \leq R[i]$ (voir Lemme 1). Ainsi $(i..R[i])$ et $(L[j]..j)$ sont des intervalles communs chevauchants. De la Proposition 1, on sait que l'intersection de deux intervalles communs chevauchants donne un intervalle commun, alors $(i..j)$ est un intervalle commun. Ainsi pour résumer, (R, L) est un générateur car pour tout i et j, $1 \leq i \leq n$, $1 \leq j \leq n$, on a :

$$R[i] \geq i \text{ et } L[j] \leq j,$$

$$(i..j) \text{ est un intervalle commun } \Rightarrow (i..j) = (i..R[i]) \cap (L[j]..j) \text{ et}$$

$$(i..j) = (i..R[i]) \cap (L[j]..j) \Rightarrow (i..j) \text{ est un intervalle commun.}$$

Ce générateur est canonique puisque pour tout $1 \leq i \leq n$, $(i..R[i])$ et $(L[i]..i)$ sont des intervalles communs.

Supposons qu'il existe un second générateur canonique (R', L'), où $R' \neq R$, alors il existe un $1 \leq i \leq n$ tel que $R'[i] < R[i]$. Puisque $(i..R[i])$ est un intervalle commun, on doit pouvoir le générer à partir de (R', L') mais $(i..R'[i]) \cap (L'[R[i]]..R[i])$ ne contient pas $R[i]$. Un raisonnement similaire s'applique si $L' \neq L$.

Finalemement, supposons deux intervalles chevauchants $(i..R[i])$ et $(j..R[j])$ où, sans perte de généralité, $i < j < R[i] < R[j]$. De la Proposition 1, on sait que l'union de deux intervalles communs chevauchants donne un intervalle commun. Alors $(i..R[j])$ est un intervalle commun, ce qui contredit l'hypothèse de maximalité de $R[i]$. Un raisonnement similaire s'applique pour L. Le générateur canonique (R, L) est donc commutant. □

Théorème 3. *Étant donné un générateur commutant (R', L'), l'Algorithme 8 calcule le générateur canonique (R, L) en temps $O(n)$.*

Algorithme 8 Le générateur canonique (R, L) étant donné un générateur commutant (R', L'). Le calcul de L est similaire et s'appuie sur le vecteur *Support* calculé selon L'

Le vecteur *Support* s'obtient de R' avec l'Algorithme 6.

$R[1] \leftarrow n$

Pour k de 2 à n **Faire**

  $R[k] \leftarrow k$

Fin Pour

Pour k de n à 2 **Faire**

  Si $(Support[k]..R[k])$ est un intervalle commun **alors**

    $R[Support[k]] \leftarrow \max(R[k], R[Support[k]])$

  Fin Si

Fin Pour

Démonstration. La complexité temporelle de l'Algorithme 8 repose sur le fait que le test permettant de déterminer si un intervalle $(i..j)$ est commun s'effectue en temps constant. En effet, de la Définition 3 de générateurs et du Lemme 1, $(i..j)$ est un intervalle commun si et seulement si $L'[j] \leq i \leq j \leq R'[i]$. Il suffit donc de tester si $L'[j] \leq i$ et $j \leq R'[i]$.

La validité de l'algorithme découle de l'observation suivante : si $R[k] \neq k$, alors il existe un entier $k' > k$ tel que

$$Support[k'] = k, \text{ et } R[k'] = R[k],$$

où $Support[k']$ (voir Section 3.3) est défini comme le plus grand entier plus petit que k' tel que $R'[Support[k']] \geq R'[k']$. Du fait que les itérations se font de façon décroissante sur k, cette observation implique que la valeur de $R[k]$ est connue au début de l'itération k.

Prouvons maintenant notre observation. On a $R[k] > k$ car, de la définition des générateurs, $R[k] \geq k$ et, de notre hypothèse, $R[k] \neq k$. De plus, $R[R[k]] = R[k]$ puisque le générateur est commutant. Ainsi, l'ensemble des entiers k'' tels que $k'' > k$ et $R[k''] = R[k]$ est non vide et contient au moins $R[k]$. Cet ensemble a donc un minimum que nous appellerons k'.

Puisque le générateur (R, L) est canonique, on a $R'[k'] \geq R[k']$ et $R'[k] \geq R[k]$. Aussi, du fait que le générateur (R', L') commute et que $k < k'$, on a $R'[k] \geq R'[k']$ (voir Figure 4.1). Alors, de la définition de *Support*, $Support[k'] \geq k$, ce qui implique que $R[Support[k']] \leq R[k]$ étant donné que les intervalles $(Support[k']..R[Support[k']])$ et $(k..R[k])$ sont commutants. Essayons maintenant de préciser les valeurs possibles de $R[Support[k']]$. Sachant que (R, L) est canonique, que $k < k'$ et que $R[k'] = R[k]$, alors $(k'..R[k'])$ et $(k..R[k])$ sont des intervalles communs chevauchants, ce qui implique que $R[Support[k']]$ ne peut être inférieur à k' (voir Proposition 1). Aussi, du fait que R commute, on ne peut avoir $k' \leq R[Support[k']] < R[k]$. La seule valeur possible de $R[Support[k']]$ est donc $R[k]$. Ainsi, afin de ne pas contredire la définition de k', on doit avoir $Support[k'] = k$. □

4.2 Le calcul des intervalles forts à partir d'un générateur canonique

Pour un ensemble $\mathcal{P}$ de K permutations, un intervalle fort est un intervalle commun qui commute avec tous les autres intervalles communs de $\mathcal{P}$. Intuitivement, l'intervalle fort revient à allonger, vers la gauche et vers la droite, toute paire A et B d'intervalles communs chevauchants de façon à obtenir $A \cup B$. De cet intervalle, on en recherche un autre qui le chevauche et si c'est le cas, on prend leur union. On répète ce processus tant qu'il y a des chevauchements et à la fin, l'intervalle étendu obtenu est nécessairement commutant avec tous les autres. On a donc un intervalle fort.

La méthode de calcul proposée s'appuie sur l'utilisation du générateur canonique de la Section 4.1 précédente, où chaque intervalle $(i..R[i])$ et $(L[j]..j)$ des vecteurs R et L est un intervalle commun. Avant de passer à la Proposition 12 démontrant la validité

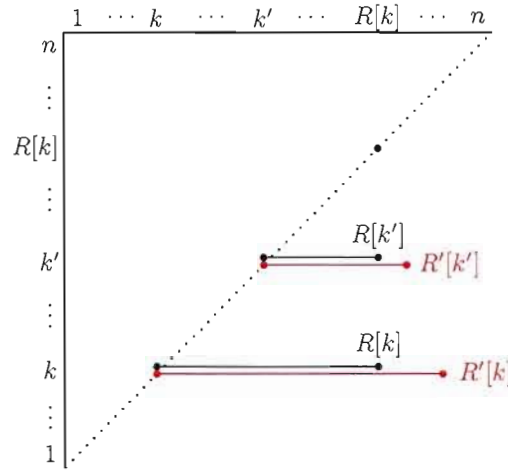


FIG. 4.1 Représentation graphique des intervalles déterminés par les vecteurs R (en noir) et R' (en rouge) du générateur canonique (R, L) et du générateur commutant (R', L'), afin d'illustrer la preuve de l'observation suivante : si $R[k] \neq k$, alors il existe un entier $k' > k$ tel que $Support[k'] = k$ et $R[k'] = R[k]$ (voir Théorème 3). Les intervalles représentés de R sont $(R[k]..R[k])$, $(k'..R[k'])$ et $(k..R[k])$ et ceux de R' sont $(k'..R'[k'])$ et $(k..R'[k])$.

de l'Algorithme 9 qui calcule les intervalles forts, nous devons passer par quatre lemmes établissant différentes propriétés liées aux générateurs canoniques.

Lemme 8. *Soit (R, L) un générateur canonique. On a :*

(1) *Si $(i..R[i])$ chevauche $(L[j]..j)$ et $(L[j']..j')$ alors $L[j] = L[j']$.*

(2) *Si $(L[j]..j))$ chevauche $(i..R[i])$ et $(i'..R[i'])$ alors $R[i] = R[i']$.*

Démonstration. (1) On suppose $L[j] \neq L[j']$ et, sans perte de généralité, $L[j] < L[j']$. Puisque $L[j] \neq L[j']$, alors $j \neq j'$ et comme L est commutant, on a $j' < j$. Par hypothèse, (R, L) est un générateur et $(i..R[i])$ chevauche $(L[j]..j)$. Alors, tel que vu au Lemme 1, $L[j] \leq i \leq j \leq R[i]$ et de la même façon. $L[j'] \leq i \leq j' \leq R[i]$. Ainsi, pour résumer, $L[j] < L[j'] \leq i \leq j' < j \leq R[i]$.

Les intervalles $(i..R[i])$ et $(L[j]..j)$ se chevauchent et puisque (R, L) est un générateur canonique, alors ces deux intervalles sont aussi communs. Ceci implique que $(L[j]..j) \setminus (i..R[i]) = (L[j]..i-1)$ est aussi un intervalle commun (voir Proposition

1). $(L[j]..i-1)$ et $(L[j']..j')$ sont des intervalles communs et ils se chevauchent car $L[j] < L[j']$ et $j' > i-1$. Alors, de la Proposition 1, $(L[j]..i-1) \cup (L[j']..j') = (L[j]..j')$ est aussi un intervalle commun, ce qui contredit la définition du générateur voulant que $L[j']$ soit l'entier minimum tel que $(L[j']..j')$ est un intervalle commun.

(2) La preuve est similaire à la preuve du Point 1. □

Considérons le générateur canonique (R, L). La fermeture transitive de la relation de chevauchement sur $R \cup L$, notée $\mathcal{G}(R, L)$, est une relation d'équivalence. Les classes d'équivalence de $\mathcal{G}(R, L)$ seront désormais appelées *classes de chevauchement*. Une classe de chevauchement triviale ne contient qu'un seul intervalle $(i..R[i])$ ou $(L[j]..j)$, qui est évidemment fort.

Les classes non triviales sont prises en compte par le Lemme 9 qui est un peu technique. Son objectif est de montrer que la structure des classes de chevauchement est très contrainte. Par exemple, une classe de chevauchement qui contiendrait six intervalles doit nécessairement ressembler à la famille d'intervalles de la Figure 4.2.

Lemme 9. *Soit $\mathcal{C}$ une classe non triviale de chevauchement contenant $(i_1..R[i_1]), \ldots, (i_k..R[i_k])$ et $(L[j_1]..j_1), \ldots, (L[j_l]..j_l)$. Supposons, sans perte de généralité, que $i_1 < \ldots < i_k$ et $j_1 < \ldots < j_l$. Alors :*

(1) *Pour tout $a, b \in (1..k)$, $R[i_a] = R[i_b]$.*

(2) *Pour tout $a, b \in (1..l)$, $L[j_a] = L[j_b]$. Ces bornes communes seront désormais notées $R[\mathcal{C}]$ et $L[\mathcal{C}]$.*

(3) *$k = l$.*

(4) *$(L[\mathcal{C}]..j_a)$ chevauche $(i_b..R[i_b])$ si et seulement si $a \geq b$.*

(5) *Pour tout $a \in (1..k-1)$, $i_{a+1} = j_a + 1$.*

(6) *$(L[\mathcal{C}]..R[\mathcal{C}])$ est un intervalle commun fort et pour tout $a \in (1..k)$, $(i_a..j_a)$ est un intervalle commun fort.*

Démonstration. **(1)** La preuve découle directement du Point 2 du Lemme 8.

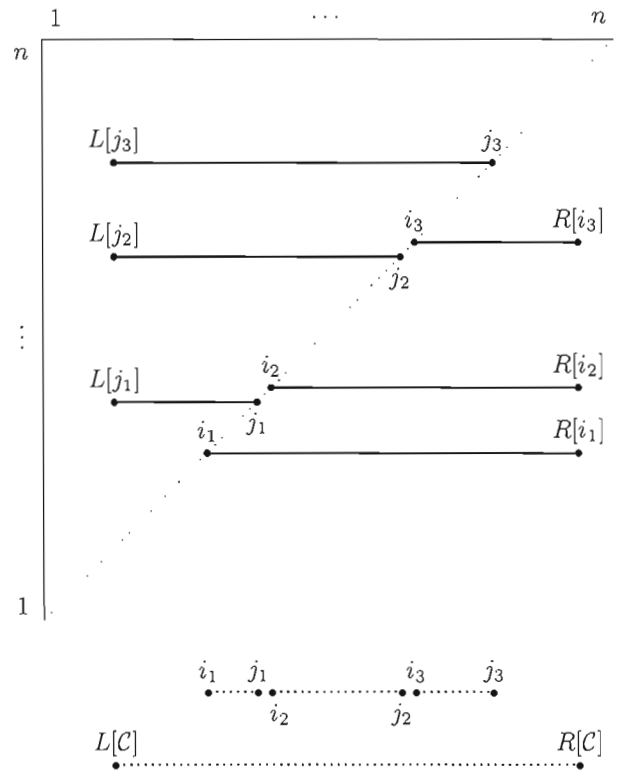


FIG. 4.2 Structure d'une classe de chevauchement comptant six intervalles communs. Ces intervalles, illustrés par des traits pleins et foncés, sont ordonnés de telle façon que $i_1 < i_2 < i_3$ et $j_1 < j_2 < j_3$. Les intervalles forts sont représentés au bas de la figure par les pointillés horizontaux. Les six points du Lemme 9 montrent les nombreuses contraintes de structure des classes de chevauchement. Dans cet exemple, on remarque que $L[j_1] = L[j_2] = L[j_3]$ et que $R[i_1] = R[i_2] = R[i_3]$ (Point 2 et 3). Il y a le même nombre d'intervalles du type $(L[j]..j)$ que du type $(i..R[i])$ soit trois (Point 3). Un intervalle $(L[j]..j)$ chevauche un intervalle $(i..R[i])$ si et seulement si $(i..R[i])$ est situé sous $(L[j]..j)$ (Point 4). On a $i_2 = j_1 + 1$ et $i_3 = j_2 + 1$ (Point 5). Enfin, les intervalles $(L[C]..R[C])$, $(i_1..j_1)$, $(i_2..j_2)$ et $(i_3..j_3)$ sont des intervalles forts (Point 6).

(**2**) La preuve découle directement du Point 1 du Lemme 8.

(**3**) Supposons $k < l$ (il existe une preuve similaire pour $k > l$). Posons f la fonction $(1..l) \rightarrow (1..k)$ telle que $f(a)$ est le plus grand b tel que $(L[\mathcal{C}]..j_a)$ chevauche $(i_b..R[\mathcal{C}])$. Puisque $k < l$, il doit exister x, y et z tel que $x \neq y$ et $f(x) = f(y) = z$. Supposons que $(i_z..R[\mathcal{C}])$ chevauche les intervalles $(L[\mathcal{C}]..j_x)$ et $(L[\mathcal{C}]..j_y)$, où, sans perte de généralité, $j_x < j_y$. (voir Figure 4.3). On a alors $L[\mathcal{C}] < i_z \leq j_x < R[\mathcal{C}]$ et $L[\mathcal{C}] < i_z \leq j_y < R[\mathcal{C}]$ qui, en intégrant l'hypothèse $j_x < j_y$, peut se résumer par $L[\mathcal{C}] < i_z \leq j_x < j_y < R[\mathcal{C}]$.

Étant donné que $(L[\mathcal{C}]..j_x)$ et $(i_z..R[\mathcal{C}])$ sont deux intervalles communs chevauchants, $(i_z..R[\mathcal{C}]) \setminus (L[\mathcal{C}]..j_x) = (j_{x+1}..R[\mathcal{C}])$ est un intervalle commun. De plus, cet intervalle chevauche $(L[\mathcal{C}]..j_y)$ car $L[\mathcal{C}] < j_{x+1} \leq j_y < R[\mathcal{C}]$.

Nous avons aussi $R[j_{x+1}] = R[\mathcal{C}]$. D'une part, si $R[j_{x+1}] < R[\mathcal{C}]$, cela contredit la maximalité de $R[j_{x+1}]$ car $(j_{x+1}..R[\mathcal{C}])$ est un intervalle commun. D'autre part, si $R[j_{x+1}] > R[\mathcal{C}]$, alors les deux intervalles communs $(j_{x+1}..R[j_{x+1}])$ et $(i_z..R[i_z])$ se chevauchent car $i_z < j_{x+1} < R[\mathcal{C}] < R[j_{x+1}]$. De la Proposition 1, ceci implique que $(j_{x+1}..R[j_{x+1}]) \cup (i_z..R[i_z]) = (i_z..R[j_{x+1}])$ est un intervalle commun, ce qui contredit la maximalité de $R[i_z] = R[\mathcal{C}]$.

(**4**) Il a été démontré au Point 3 que la fonction f est bijective où, pour toute paire d'intervalles $(L[\mathcal{C}]..j_x)$ et $(L[\mathcal{C}]..j_y)$, $j_x < j_y$, il existe un intervalle $(i = j_x+1..R[\mathcal{C}])$ qui chevauche $(L[\mathcal{C}]..j_y)$. On a donc une alternance d'intervalles chevauchants $(L[\mathcal{C}]..j)$ et $(i..R[\mathcal{C}])$. Puisque $i_1 < \ldots < i_k$ et $j_1 < \ldots < j_l$, la fonction f est croissante et forcément, $f(a) = a$. Cela signifie que a est le plus grand b tel que $(L[\mathcal{C}]..j_a)$ chevauche $(i_b..R[\mathcal{C}])$. On a alors $L[\mathcal{C}] < i_b \leq j_a < R[\mathcal{C}]$ qui reste vrai pour tout $i \leq i_b$. Puisque $i_1 < \ldots < i_k$ et $j_1 < \ldots < j_l$, on obtient que $(L[\mathcal{C}]..j_a)$ chevauche $(i_b..R[\mathcal{C}])$ pour tout $b \leq a$.

(**5**) Du Point 4, pour tout $a \in (1..k-1)$, $(L[\mathcal{C}]..j_a)$ chevauche $(i_a..R[\mathcal{C}])$ mais ne chevauche pas $(i_{a+1}..R[\mathcal{C}])$ qui est chevauché par $(L[\mathcal{C}]..j_{a+1})$. Comme $(L[\mathcal{C}]..j_a)$ et $(i_a..R[\mathcal{C}])$ sont des intervalles communs chevauchants, le deuxième moins le premier donne l'intervalle commun $(j_a + 1..R[\mathcal{C}])$. De plus, $R[j_a + 1] = R[\mathcal{C}]$ car

si $R[j_a + 1] < R[\mathcal{C}]$, $(j_a + 1..R[\mathcal{C}])$ ne pourrait être un intervalle commun et si $R[j_a + 1] > R[\mathcal{C}]$, $R[i_a] = R[\mathcal{C}]$ ne serait pas l'entier maximum tel que $(i_a..R[i_a])$ est un intervalle commun.

Du chevauchement de $(L[\mathcal{C}]..j_a)$ et $(i_a..R[\mathcal{C}])$, on a $L[\mathcal{C}] < i_a \leq j_a < R[\mathcal{C}]$ et $L[\mathcal{C}] < j_a < j_a + 1$. De l'hypothèse $j_1 < \ldots < j_l$ on a $j_a + 1 \leq j_{a+1}$. Finalement, du fait que les intervalles $(L[\mathcal{C}]..j_{a+1})$ et $(i_{a+1}..R[\mathcal{C}])$ se chevauchent, nous avons $L[\mathcal{C}] < i_{a+1} \leq j_{a+1} < R[\mathcal{C}]$. Alors, pour résumer, $L[\mathcal{C}] < j_a+1 \leq j_{a+1} < R[\mathcal{C}]$ ce qui implique le chevauchement des intervalles communs $(j_a + 1..R[\mathcal{C}])$ et $(L[\mathcal{C}]..j_{a+1})$ et ainsi, $(j_a + 1..R[\mathcal{C}]) \in \mathcal{C}$.

Puisqu'il a été démontré que f est une fonction bijective où $f(a) = a$ (voir Point 4) et que, par hypothèse, $i_1 < \ldots < i_k$, la valeur i_{a+1} qui succède à i_a doit être le plus petit entier supérieur à j_a tel que $(i_{a+1}..R[\mathcal{C}])$ chevauche $(L[\mathcal{C}]..j_{a+1})$. Cet entier est $j_a + 1$ et alors $i_{a+1} = j_a + 1$. En effet, si $i_{a+1} \leq j_a$, on aurait $(L[\mathcal{C}]..j_a)$ chevauchant $(i_{a+1}..R[\mathcal{C}])$ et $f(a) = a + 1$, contredisant ainsi le Point 4.

(6) Par hypothèse, $(L[\mathcal{C}]..j_1)$ et $(i_1..R[\mathcal{C}])$ sont deux intervalles communs et du Point 4, ils sont de plus chevauchants. Alors leur union $(L[\mathcal{C}]..R[\mathcal{C}])$ est aussi un intervalle commun. Similairement, $(L[\mathcal{C}]..j_a)$ et $(i_a..R[\mathcal{C}])$ sont deux intervalles communs et du Point 4, ils sont chevauchants. Alors leur intersection $(i_a..j_a)$ est aussi un intervalle commun.

Supposons un intervalle commun $(i..j)$ qui chevauche $(L[\mathcal{C}]..R[\mathcal{C}])$ et, sans perte de généralité, $i < L[\mathcal{C}] \leq j < R[\mathcal{C}]$. Puisque $(i..j)$ est un intervalle commun, on a $L[j] \leq i \leq j \leq R[i]$ et ainsi $L_j < L[\mathcal{C}]$. On a aussi $i_1 > L[\mathcal{C}]$ car $(L[\mathcal{C}]..j_1)$ et $(i_1..R[\mathcal{C}])$ sont deux intervalles communs chevauchants. Alors on a $L[j] < L[\mathcal{C}] < i_1 \leq j < R[\mathcal{C}]$, ce qui implique que $(i_1..R[i_1] = R[\mathcal{C}]$ et $(L[j]..j)$ sont chevauchants. Du Point 4, $(i_1..R[i_1]$ chevauche aussi $(L[j_1]..j_1)$. Dans ce cas, du Point 1 du Lemme 8, on doit avoir $(L[j_1] = (L[j] = L[\mathcal{C}]$ ce qui contredit le fait que sous notre hypothèse, $L_j < L[\mathcal{C}]$. Il ne peut donc exister d'intervalle commun chevauchant l'intervalle $(L[\mathcal{C}]..R[\mathcal{C}])$.

Supposons un intervalle commun $(i..j)$ qui chevauche $(i_a..j_a)$ et, sans perte de

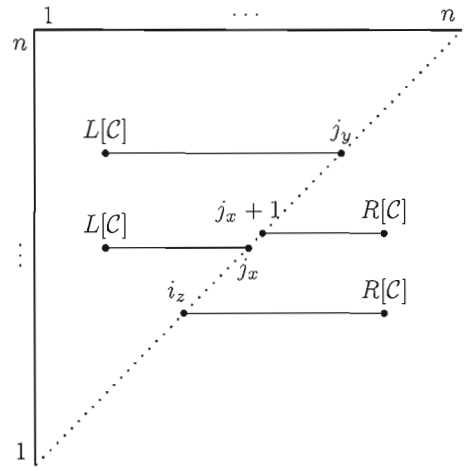


FIG. 4.3 Si, pour une classe de chevauchement $\mathcal{C}$ donnée, on a les intervalles $(L[\mathcal{C}]..j_x)$ et $(L[\mathcal{C}]..j_y)$, $j_x < j_y$, il existe un intervalle $(i = j_x + 1..R[\mathcal{C}]) \in \mathcal{C}$ situé entre les deux. (voir Point 3 de la démontration du Lemme 9).

généralité, $i < i_a \leq j < j_a$. Puisque $(i_a..j_a)$ vient de l'intersection des intervalles communs chevauchants $(i_a..R[\mathcal{C}])$ et $(L[\mathcal{C}] = L[j_a]..j_a)$, on a $j_a < R[\mathcal{C}]$. Ainsi $i < i_a \leq j < R[\mathcal{C}]$, ce qui implique que les intervalles communs $(i_a..R[\mathcal{C}])$ et $(i..j)$ sont chevauchants et que leur union $(i..R[\mathcal{C}])$ est aussi un intervalle commun. Cependant, par hypothèse, on a $i < j_a$ et alors, du Point 4, $(i..R[\mathcal{C}])$ chevauche $(L[\mathcal{C}]..j_a)$. Ainsi $(i..R[\mathcal{C}])$ doit appartenir à la classe chevauchante $\mathcal{C}$, ce qui vient en contradiction avec le fait que $i \notin \{i_1..i_k\}$. Il ne peut donc exister d'intervalle commun chevauchant un intervalle $(i_a..j_a)$.

□

Notons que si une classe de chevauchement $\mathcal{C}$ n'est pas triviale, alors $(L[\mathcal{C}]..R[\mathcal{C}])$ n'est pas un intervalle de (R, L). Ces intervalles forment cependant une famille commutante comme le prouve le lemme suivant. De plus, d'ici la fin du présent travail, la borne d'un intervalle (voir Section 1.2) fera maintenant référence à un élément plutôt qu'à une position.

Lemme 10. *Soit $\mathcal{C}$ et $\mathcal{C}'$, deux classes de chevauchement de $\mathcal{G}(R, L)$. Alors* $(L[\mathcal{C}]..R[\mathcal{C}])$

et$(L[\mathcal{C}']..R[\mathcal{C}'])$ *commutent.*

Démonstration. Si $\mathcal{C}$ est une classe triviale, posons $I = J$ l'unique membre de cette classe, sinon, I et J sont deux intervalles chevauchants. Des Points 3 et 4 du Lemme 9, $L[\mathcal{C}]$ est la borne gauche d'un de ces intervalles et $R[\mathcal{C}]$ la borne droite de l'autre. On suppose, sans perte de généralité, $L[\mathcal{C}]$ la borne gauche de I et $R[\mathcal{C}]$ la borne droite de J. Soit I' un intervalle de $\mathcal{C}'$ où, naturellement, $I' \subseteq (L[\mathcal{C}']..R[\mathcal{C}'])$. Trois cas sont alors possibles.

1. Soit que $I \subset I'$ et $J \subset I'$. Alors $I \cup J = (L[\mathcal{C}]..R[\mathcal{C}]) \subset I'$ ce qui implique que $(L[\mathcal{C}]..R[\mathcal{C}]) \subset (L[\mathcal{C}']..R[\mathcal{C}'])$.
2. Soit que $I' \subset I$ et $I' \subset J$. Si un intervalle J' chevauche I', alors $J' \subset I$ et $J' \subset J$ car autrement on aurait $J' \in \mathcal{C}$, une contradiction. Si I' est le seul intervalle de $\mathcal{C}$, alors $I' = J'$. Ainsi $I' \cup J' = (L[\mathcal{C}']..R[\mathcal{C}']) \subset I \cup J = (L[\mathcal{C}]..R[\mathcal{C}])$.
3. Soit que $I \cap I' = \varnothing$ et $J \cap I' = \varnothing$. Alors $I \cup J = (L[\cap C]..R[\mathcal{C}]) \cap I' = \varnothing$ ce qui implique que $(L[\mathcal{C}]..R[\mathcal{C}]) \subset (L[\mathcal{C}']..R[\mathcal{C}']) = \varnothing$.

□

Le lemme suivant montre que tous les intervalles forts sont obtenus grâce à la construction du Point 6 du Lemme 9.

Lemme 11. *Soit S un intervalle fort. Il existe une classe de chevauchement $\mathcal{C}$ telle que, soit $S = (L[\mathcal{C}]..R[\mathcal{C}])$, ou soit $\mathcal{C}$ contienne k intervalles de R, $(i_1..R[i_1]), \ldots, (i_k..R[i_k]))$, $i_1 < \ldots < i_k$, et k intervalles de L, $(L[j_1]..j_1), \ldots, (L[j_k]..j_k]))$, $j_1 < \ldots < j_l$ où il existe un $a \in (1..k)$ tel que $S = (i_a..j_a)$.*

Démonstration. Posons $F = (i..j)$ un intervalle fort. Si $R[i] = j$ alors $F \in R$ et $F = (L[\mathcal{C}]..R[\mathcal{C}])$ forme une classe de chevauchement triviale. Un raisonnement similaire s'applique si $L[j] = i$. Sinon, $(i..R[i])$ et $(L[j]..j)$ se chevauchent et appartiennent à une classe $\mathcal{C}$ non triviale ce qui implique que pour une certaine valeur de $a, b \in (1..k)$, nous avons $i = i_a$ et $j = j_b$. Du Point 4 du Lemme 9, si $a > b$, alors $(L[\mathcal{C}]..j_b)$ ne peut

chevaucher $(i_a..R[i_a])$ et si $a < b$, alors F est chevauché par $(i_b..R[\mathcal{C}])$, ce qui contredit l'hypothèse voulant que F soit fort. On a donc forcément $a = b$. □

Soit (R, L) un générateur canonique. Considérons les $4n$ bornes des intervalles des familles $(i..R[i])$ et $(L[j]..j)$, $i, j \in (1..n)$. Soit $(a_1, \ldots, a_{4n})$ la liste de ces $4n$ bornes triées en ordre croissant, où les bornes de gauches sont placées avant les bornes de droite, lorsqu'elles sont égales. Par exemple, pour l'ensemble de permutations $\mathcal{P} = \{Id_{10}, P\}$ où $P = (3\ 2\ 1\ 8\ 9\ 10\ 7\ 6\ 5\ 4)$, on a le générateur canonique (R, L) suivant :

$R = \{(1..10), (2..3), (3..3), (4..10), (5..10), (6..10), (7..10), (8..10), (9..10), (10..10)\}$ et

$L = \{(1..1), (1..2), (1..3), (4..4), (4..5), (4..6), (4..7), (8..8), (8..9), (1..10)\}$,

à partir duquel on obtient la liste ordonnée suivante :

(1,1,1,1,1,$\overline{1}$,2,$\overline{2}$,3,$\overline{3}$,$\overline{3}$,$\overline{3}$,4,4,4,4,4,$\overline{4}$,5,$\overline{5}$,6,$\overline{6}$,7,$\overline{7}$,8,8,8,$\overline{8}$,9,$\overline{9}$,10,$\overline{10}$,$\overline{10}$,$\overline{10}$,$\overline{10}$,$\overline{10}$,$\overline{10}$,$\overline{10}$,$\overline{10}$,$\overline{10}$)

où $\overline{i}$ signale une borne droite. On peut voir à la Figure 4.4a la représentation graphique de ce générateur.

Cette liste peut être facilement construite en temps linéaire. Il s'agit d'effectuer un parcours des deux vecteurs R et L, où l'on note pour chaque $i \in (1..n)$ s'il est une borne gauche, ce qui arrive au moins une fois, ou une borne droite, ce qui se produit aussi au moins une fois.

Ce simple algorithme parcourt de gauche à droite la liste des bornes en empilant les bornes gauches. Si une borne droite est rencontrée, un intervalle fort est généré. Ainsi, l'ordre de production des intervalles suivra l'ordre croissant des bornes droites de la liste. Tous les intervalles du type $(i_a..j_a)$ d'une classe de chevauchement donnée seront générés avant que ne le soient ceux du type $(L[\mathcal{C}]..R[\mathcal{C}])$, car $j_a \leq R[\mathcal{C}]$. Si une classe $\mathcal{C}_1$ de chevauchement est incluse dans une classe $\mathcal{C}_2$, alors $\mathcal{C}_1$ verra tous ses intervalles générés avant $\mathcal{C}_2$ puisque $R[\mathcal{C}_1] < R[\mathcal{C}_2]$.

Algorithme 9 Calcul des intervalles communs forts

S est une pile de bornes et s le dessus de cette pile.

Pour i de 1 à $4n$ **Faire**

  Si a_i est une borne gauche **alors**

    Empiler a_i dans S

  Sinon

    Générer l'intervalle fort $(s..a_i)$

    Dépiler S

  Fin Si

Fin Pour

Traçons brièvement l'algorithme à l'aide du générateur de l'exemple ci-dessus, illustré à la Figure 4.4a. Les cinq premières bornes, valant chacune 1, sont empliées puis à la sixième, qui est une borne droite, on produit l'intervalle (1..1) et on dépile. La prochaine borne de la liste est 2 que l'on empile et la suivante, qui est aussi un 2, provoque la sortie de l'intervalle (2..2) suivi d'un dépilement. Le processus est similaire pour produire (3..3) et après le dépilement de la borne gauche 3, la pile ne contient que des 1. Les deux bornes suivantes de la liste valent 3 et sont des bornes droites, ce qui entraîne deux fois la sortie de l'intervalle (1..3). En suivant cette trace à l'aide de la Figure 4.4a, on comprend intuitivement que cette méthode permet de générer tous les intervalles $(i_a..j_a)$ et $(L[\mathcal{C}]..R[\mathcal{C}])$, soit tous intervalles forts. Démontrons maintenant cet algorithme de façon formelle.

Proposition 12. *Étant donné la liste ordonnée* $(a_1, \ldots, a_{4n})$ *des* $4n$ *bornes d'un générateur canonique* (R, L). *l'Algorithme 9 génère les intervalles forts de* $\mathcal{G}(R, L)$ *en temps* $O(n)$.

Démonstration. La complexité temporelle est évidente. Prouvons sa validité. Nous devons prouver dans un premier temps que chaque intervalle généré par l'algorithme est fort. Du Lemme 10, on sait que les différentes classes $\mathcal{C}$ de chevauchement commutent entre elles. De l'ordonnancement de la liste de bornes et du fait que l'algorithme empile

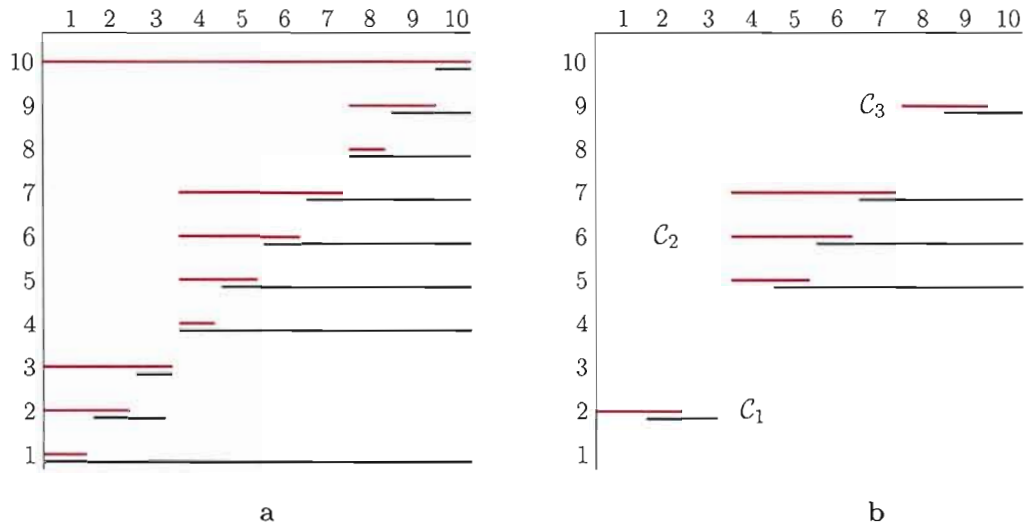


FIG. 4.4 **a)** Représentation du générateur canonique (R, L) calculé pour l'ensemble de permutations $\mathcal{P} = \{Id_{10}, P\}$, où $P = (3\ 2\ 1\ 8\ 9\ 10\ 7\ 6\ 5\ 4)$. Les intervalles du vecteur L sont en rouge et celles du vecteur R sont en noir. **b)** Les trois classes chevauchantes non triviales de $\mathcal{G}(R, L)$, la fermeture transitive de la relation de chevauchement sur $R \cup L$.

chaque borne gauche rencontrée, à tout moment, le dessus de la pile contient les bornes gauches d'une classe $\mathcal{C}$ de chevauchement. Le reste de la pile contient les bornes gauches des autres classes de chevauchement qui contiennent $\mathcal{C}$, par ordre d'inclusion, où les bornes de la classe la plus inclusive sont situées au fond de la pile. Ainsi, chaque classe incluse dans une autre verra tous ses intervalles générés avant de compléter la classe englobante suivante.

Selon le Point 3 du Lemme 9, chaque classe de chevauchement contient le même nombre d'intervalles $(L[j_a]..j_a)$ et $(i_b..R[i_b])$. Alors, étant donné l'ordonnancement des bornes de la liste et du fait que les classes commutent, l'algorithme ne génère que les intervalles dont les bornes appartiennent à une même classe de chevauchement.

Si une classe de chevauchement est triviale, clairement, seul son intervalle sera généré. On suppose maintenant que cette classe n'est pas triviale. Nous allons démontrer que les intervalles générés sont exactement les $(i_a..j_a)$ et $(L[\mathcal{C}]..R[\mathcal{C}])$ décrits au Point 6 du Lemme 9. Il s'agit d'observer qu'il y a k empilements de bornes $L[\mathcal{C}]$ nécessairement suivi d'un empilement de la borne i_1 de l'intervalle $(i_1..j_1)$. Pour chaque $a \in (1..k-1)$, les trois instructions suivantes sont alors répétées : génération de $(i_a..j_a)$, dépilement de i_a puis empilement de la borne i_{a+1}. En terminant, il y a génération de $(i_k..j_k)$, dépilement du i_k, puis en tout k dépilements, où chaque dépilement est effectué après avoir généré chacun des k intervalles $(L[\mathcal{C}]..R[\mathcal{C}])$. Ainsi l'algorithme a produit les k différents intervalles $(i_a..j_a)$ et les k intervalles $(L[\mathcal{C}]..R[\mathcal{C}])$ de la classe $\mathcal{C}$, soit uniquement des intervalles forts.

Dans un deuxième temps, on sait du Lemme 11 que les intervalles forts d'une classe $\mathcal{C}$ sont du type $(i_a..j_a)$ ou $(L[\mathcal{C}]..R[\mathcal{C}])$. Puisqu'ils sont tous générés par l'algorithme, la preuve est complète. □

Il est utile de noter que l'Algorithme 9 produit invariablement $2n$ intervalles forts. Sachant que le nombre de ces intervalles est situé entre $n+1$ et $2n-1$, certains intervalles seront forcément répétés plus d'une fois.

4.3 Représentation des intervalles forts

Maintenant que les intervalles forts sont identifiés, nous sommes en mesure des les représenter graphiquement. Considérons l'ensemble de permutations $\mathcal{P} = \{Id_n, P\}$. Les intervalles forts de $\mathcal{P}$ sont partiellement ordonnés par la relation d'inclusion et cet ordre d'inclusion permet de définir un arbre tel que :

- chaque noeud est un intervalle fort de l'ensemble $\mathcal{P}$,
- la racine contient la permutation P,
- chacune des n feuilles est constituée d'un élément différent de P,
- la lecture des feuilles de gauche à droite correspond à P,
- la relation parent-enfant des noeuds traduit la relation d'inclusion des intervalles.

La construction d'un arbre pour un ensemble de K permutations est possible à l'aide de l'arbre PQ de (Booth et Lueker, 1976). Sa construction s'effectue en temps linéaire par l'Algorithme 7 de (Bergeron et al., 2005) qui ne sera pas discuté dans le présent travail. Nous nous limiterons ici à la présentation de l'arbre des intervalles forts d'un ensemble de deux permutations (dont Id_n) et à discuter des avantages que procure l'utilisation les intervalles forts par rapport aux intervalles communs ordinaires.

La Figure 4.5, tirée des travaux de (Gingras et al. 2005), illustre l'arbre des intervalles forts pour le chromosome X du rat et de la souris. Clairement, on note que les intervalles forts triviaux sont situés à la racine et aux feuilles tandis que les intervalles non triviaux sont situés dans les noeuds internes, excepté la racine. L'intérêt d'une représentation par un arbre réside dans la facilité avec laquelle on peut visualiser directement les structures conservées et la façon dont elles sont imbriquées. Cette représentation intuitive est rendue possible par le fait que le nombre d'intervalles forts est linéaire, comparativement au nombre quadratique d'intervalles communs.

Par exemple, si on reprend les mêmes données de la Figure 4.5, une représentation exhaustive des intervalles communs donnerait les 43 intervalles communs suivants :

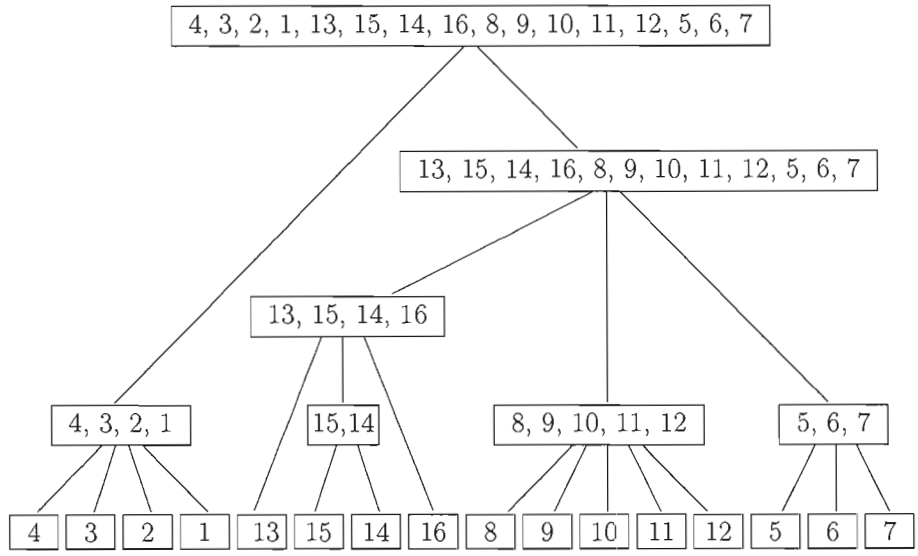


FIG. 4.5 L'arbre d'inclusion des intervalles forts du chromosome X de la souris et du rat. Chaque noeud représente un intervalle fort. Les nombres sont les étiquettes de 16 longs segments communs d'ADN. Ces étiquettes, ordonnées chez la souris, sont modélisées par la permutation identité Id_{16}. L'autre permutation modélise l'ordre de ces même segments chez le rat. La lecture des feuilles de gauche à droite de même que la racine représentent la séquence complète du rat (Gingras et al., 2005).

{(1..1), (2..2), (1..2), (3..3), (2..3), (1..3), (4..4), (3..4), (2..4), (1..4), (5..5), (6..6), (5..6), (7..7), (6..7), (5..7), (8..8), (9..9), (8..9), (10..10), (9..10), (8..10), (11..11), (10..11), (9..11), (8..11), (12..12), (11..12), (10..12), (9..12), (8..12), (5..12), (13..13), (14..14), (15..15), (14..15), (13..15), (16..16), (14..16), (13..16), (8..16), (5..16), (1..16)},

tandis qu'une représentation à l'aide d'un générateur donnerait les deux vecteurs suivants :

$$Inf[1,1,1,1,5,5,5,8,8,8,8,5,13,14,13,1]$$

$$Sup[16,4,4,4,16,7,7,16,12,12,12,12,16,16,15,16]$$

Bien que très compacte, cette dernière représentation ne permet pas une appréciation rapide et directe des groupes conservés tandis que la liste exhaustive des intervalles communs embrouille le lecteur par une surabondance d'intervalles. Outre les $n+1$ intervalles triviaux, on y retrouve beaucoup d'intervalles dont la contribution informative est faible. Par exemple, prenons l'intervalle commun fort (1..4) du précédent jeu de données. Sachant que le groupe conservé (1..4) existe, l'identification des intervalles communs (1..2), (2..3), (1..3), (3..4) et (2..4) devient peu significative.

L'élimination des intervalles faibles ne compromet donc pas l'atteinte de notre but qui est, rappelons-nous, l'obtention d'une méthode pratique et efficace pour l'identification des groupes conservés entre K génomes modélisés par autant de permutations. Ainsi, l'usage des intervalles forts permet d'atteindre ce but avec plus de précision et ouvre la voie à une représentation graphique intuitive de ces groupes, ce qui facilite leur interprétation.

CONCLUSION

Le problème de l'identification de groupes conservés d'ADN entre K différentes espèces se modélise par le biais de K permutations, où chaque entier est une étiquette représentant un gène ou un segment d'ADN. La résolution de ce problème nécessite des algorithmes efficaces en pratique étant donné les tailles importantes, et en très forte croissance, des données biologiques utilisées en entrée.

La notion d'intervalles communs de Uno et Yagiura (2000) formalise l'idée des groupes conservés. Nous avons d'abord survolé les premières méthodes de calcul qui, malgré une complexité temporelle optimale, demeurent lourdes à implémenter et peu efficaces en pratique. Nous avons ensuite présenté la méthode de Bergeron et al. (2005) qui propose une solution largement simplifiée, tant au niveau des algorithmes que des structures de données, et qui permet un maximum d'efficacité et de facilité d'implémentation. Leur méthode, basée sur les générateurs, permet aussi une représentation simple, dans un espace linéaire, du nombre quadratique d'intervalles communs. Par la suite, nous avons présenté une nouvelle méthode, dite de *projection*, qui généralise certains algorithmes de Bergeron et al. (2005), tout en conservant la même simplicité et efficacité pratique.

En dernière partie, nous avons présenté la notion d'intervalles communs forts et l'algorithme de Bergeron et al. (2005) qui en effectue le calcul, aussi de façon simple et efficace, grâce à l'utilisation du générateur canonique. Nous avons finalement montré que les intervalles communs forts sont plus efficaces à identifier les groupes conservés et que, contrairement aux intervalles communs, ils permettent une représentation graphique sous forme d'arbre, ce qui facilite une analyse intuitive.

Nous avons présenté, entre autres, les meilleurs algorithmes qui permettent de calculer les intervalles communs et les intervalles forts, pour un ensemble de K permutations. Étant donné leur extrême simplicité et leur grande efficacité, autant théorique que pratique, nous croyons peu vraisemblable qu'il soit possible d'améliorer significativement leur performance.

BIBLIOGRAPHIE

(1) A. Bergeron, C. Chauve, F. de Montgolfier et M. Raffinot. *Computing common intervals of K permutations, with applications to modular decomposition of graphs*, European Symposium on Algorithms (ESA 2005), Lecture Notes in Computer Science vol. 3669, pp 779-790, Springer-Verlag, Berlin, 2005.

(2) A. Bergeron et J. Stoye. *On the similarity of sets of permutations and its applications to genome comparison*, 9th Annual International Conference on Computing and Combinatorics (COCOON 2003), Lecture Notes in Computer Science vol. 269, pp 68-79, Springer-Verlag, Berlin, 2003.

(3) S. Bérard, A. Bergeron, C. Chauve et C. Paul. *Perfect sorting by reversals is not always difficult*, Lecture Notes in Computer Science vol. 3692, pp 228-238, Springer-Verlag, Berlin, 2005.

(4) S. Booth et S. Lueker. *Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms*, J. Comput. Syst. Sci., 13, pp 335-379, 1976.

(5) B. M. Bui Xuan, M. Habib, et C. Paul. *Revisiting T. Uno and M. Yagiura's Algorithm*, Lecture Notes in Computer Science vol. 3827, pp 146-155, Springer-Verlag, Berlin, 2005.

(6) F. de Montgolfier. *Décomposition modulaire des graphes. Théorie, extensions et algorithmes*, Thèse soutenue le 5 décembre 2003 à Montpellier à l'Université des Sciences et Techniques du Languedoc.

(7) O.Gingras, Y.Gingras, A. Levasseur, A. Bergeron et Cedric Chauve. *A software tool for whole genome synteny comparisons*, Poster à Research in Computational Molecular Biology (RECOMB 2005).

(8) S. Heber et J. Stoye. *Finding all common intervals of k permutations*, Combinatorial Pattern Matching, 12th Annual Symposium (CPM 2001), Lecture Notes in Computer Science vol. 2089, pp 207–218, Springer-Verlag, 2001.

(9) G.M. Landau, L. Parida et O. Weimann. *Gene Proximity Analysis Across Whole Genomes via PQ Trees*, Journal of Computational Biology vol. 12, num. 10, pp. 1289–1306, 2005.

(10) T. Marquès-Bonet, M. Cáceres, J. Bertranpetit, T. M. Preuss, J. W. Thomas et A. Navarro. *Chromosomal rearrangements and the genomic distribution of gene-expression divergence in humans and chimpanzees*, Trends in Genetics

Volume 20, Issue 11 , November 2004, Pages 524-529.

(11) NCBI News, *GenBank® Passes the 100 Gigabase Mark.* NCBI News (http ://www.ncbi.nlm.nih.gov/About/newsletter.html), Vol 14, No 2, Nov 2005.

(12) The Chimpanzee Sequencing and Analysis Consortium. *Initial sequence of the chimpanzee genome and comparison with the human genome*, Nature 437, 69-87, 1 September 2005.

(13) T. Uno et M. Yagiura. *Fast algorithms to enumerate all common intervals of two permutations*, Algorithmica, 26(2) :290–309. 2000.

(14) M. Yagiura, H. Nagamochi et T. Ibaraki. *Two Comments on the Subtour Crossover Exchange Operator* (en japonais), Technical Report of IEICE (COMP94-18) 94, No. 88, pp 1-10, 1994. (Version abrégée dans Journal of Japanese Society for Artificial Intelligence, 10, 464-467, 1995.)

ANNEXE A

CODE SOURCE JAVA

Annexe A code source java

```
public class MainCommonIntervals {

    //constants
    static final int ASCENDING = 1;
    static final int DESCENDING = -1;

    //widened permutation to be used. First and last indexes are for internal use only
    int[] p = {0, 4, 3, 2, 1, 13, 15, 14, 16, 8, 9, 10, 11, 12, 5, 6, 7, 0};

    //data structures
    int n; // nb of elements in narrow p (without 0 and n indexes)
    int[] LMax, RMax, RMin, LMin; // L & R bounds of IMax and IMin.
                                  // First and last index are for internal use only
    int[] Inf, Sup; // The generator.
    int[] Rcanonical, Lcanonical; // The canonical generator.
    int[] supportR, supportL; //
    Bound[] boundArray; //Rcanonical and Lcanonical 4n bounds array.
    int[] pInverse; // inversed (not reversed) perm p



    public static void main(String[] args) {
        MainCommonIntervals mainCommonIntervals = new MainCommonIntervals ();
    }



    public MainCommonIntervals() {
        //inits of data structures
        n = p.length - 2; // n is the greatest perm element
        LMax = new int[p.length]; //by positions
        RMax = new int[p.length]; //by positions
        LMin = new int[p.length]; //by positions
        RMin = new int[p.length]; //by positions

        Inf = new int[p.length];
        Sup = new int[p.length];
        Rcanonical = new int[p.length];
        Lcanonical = new int[p.length];
        supportR = new int[p.length];
        supportL = new int[p.length];
        pInverse = inverse();

        System.out.println();
        showNarrowIntArray("perm:", p);
        showNarrowIntArray("pInverse:", pInverse);

        //computing intervals
        computeGenerator();
        computeSupportR();
        generateCommonIntervals();
        computeCanonicalGenerator();
        build4nBoundList();
        computeStrongIntervals();
    }
```

```
// Using projection strategy
private void computeGenerator() {
    int[] Ideb = new int[p.length];
    int[] Ifin = new int[p.length];
    int[] Idn = makeIdn(n);

    /////////////////
    // Computing Inf
    /////////////////

    // init interval families {(1..i)}
    for (int i = 1; i < p.length - 1; i++) {
        Ideb[i] = 1;
        Ifin[i] = i;
    }

    //First projection
    projection(Ideb, Ifin, Idn, p, 1, LMin, RMin);

    //Second projection
    int[] dummy = new int[p.length];
    projection(LMin, RMin, p, Idn, 1, Inf, dummy);

    showNarrowIntArray("Inf:", Inf);

    /////////////////
    // Computing Sup
    /////////////////

    // init interval families {(i..n)}
    for (int i = n; i >= 1; i--) {
        Ideb[i] = i;
        Ifin[i] = n;
    }

    //First projection
    projection(Ideb, Ifin, Idn, p, -1, LMax, RMax);

    //Second projection
    projection(LMax, RMax, p, Idn, -1, dummy, Sup);

    showNarrowIntArray("Sup:", Sup);
}
```

```
public void projection(int[] Ideb, int[] Ifin, int[] P, int[] Q, int direction,
                       int[] Jdeb, int[] Jfin) {
    // PRE: Only 2 forms of {I(i)} are valid
    // PRE: If i is the greatest elem of I(i), {I(i)} = [(1..1), ..., (1..n)]
    // PRE: If i is the smallest elem of I(i), {I(i)} = [(n..n), ..., (1..n)]
    int[] indexes = new int[n + 1];
    int[] Pinv = inverse(P);
    int[] Qinv = inverse(Q);

    // inits
    for (int i = 1; i < n + 1; i++) {
        Jdeb[i] = Qinv[i];
        Jfin[i] = Qinv[i];
    }

    //setting indexes to follow
    if (direction == ASCENDING) {
        for (int i = 1; i <= n; i++)
            indexes[i] = i;
    }
    else {
        for (int i = 1; i <= n; i++)
            indexes[i] = n - i + 1;
    }

    // computing the projection
    for (int i = 1; i <= n; i++) {
        int k = indexes[i];
        // Expand to the left
        while (Pinv[Q[Jdeb[k] - 1]] >= Ideb[k] && Pinv[Q[Jdeb[k] - 1]] <= Ifin[k])
            Jdeb[k] = Jdeb[Q[Jdeb[k] - 1]];

        // Expand to the right
        while (Pinv[Q[Jfin[k] + 1]] >= Ideb[k] && Pinv[Q[Jfin[k] + 1]] <= Ifin[k])
            Jfin[k] = Jfin[Q[Jfin[k] + 1]];
    }
}
```

```
private void computeSupportR() {
        Stack<Integer> S = new Stack<Integer> ();
        S.push(1);
        for (int i = 2; i <= n; i++) {
            while (Sup[S.peek()] < i)
                S.pop();
            supportR[i] = S.peek();
            S.push(i);

        }

        System.out.println();
        showNarrowIntArray("supportR:", supportR);
    }



    private void computeSupportL() {
        Stack<Integer> S = new Stack<Integer> ();
        S.push(n);
        for (int i = n - 1; i >= 1; i--) {
            while (Inf[S.peek()] > i)
                S.pop();
            supportL[i] = S.peek();
            S.push(i);
        }

        System.out.println();
        showNarrowIntArray("supportL:", supportL);
    }



    private void generateCommonIntervals() {
        System.out.println();
        System.out.println("Common Intervals");
        int i;
        for (int j = n; j >= 1; j--) {
            System.out.print("j=" + j + "  ");
            i = j;

            while (i >= Inf[j]) {
                System.out.print("(" + i + ".." + j + ") ");
                i = supportR[i];
            }
            System.out.println();
        }
    }
```

```
//PRE: vectors supportR, Inf and Sup are already computed
private void computeCanonicalGenerator() {
    //computing Rcanonical
    //init
    Rcanonical[1] = n;
    for (int k = 2; k <= n; k++)
        Rcanonical[k] = k;
    //computing
    for (int k = n; k >= 1; k--) {
        if (Inf[Rcanonical[k]] <= supportR[k] && Rcanonical[k] <= Sup[supportR[k]])
            Rcanonical[supportR[k]] = Math.max(Rcanonical[k], Rcanonical[supportR[k]]);
    }

    //computing Lcanonical
    computeSupportL();
    //init
    Lcanonical[n] = 1;
    for (int k = 1; k < n; k++)
        Lcanonical[k] = k;
    //computing
    for (int k = 1; k < n; k++) {
        if (Inf[Lcanonical[k]] <= supportL[k] && Lcanonical[k] <= Sup[supportL[k]])
            Lcanonical[supportL[k]] = Math.min(Lcanonical[k], Lcanonical[supportL[k]]);
    }

    showIntArray("Rcanonical", Rcanonical, 1, n);
    showIntArray("Lcanonical", Lcanonical, 1, n);
}
```

```
private void computeStrongIntervals() {
    System.out.println();
    System.out.println("Strong Intervals");
    Stack<Integer> S = new Stack<Integer> ();
    for (int i = 1; i < boundArray.length; i++) {
        if (boundArray[i].isLeftBound())
            S.push(boundArray[i].getValue());
        else {
            System.out.println("(" + S.peek() + ".." + boundArray[i].getValue() + ")");
            S.pop();
        }
    }
}
```

```
//From the CanonicalGenerator,
//it builds an array of the 4n bounds of family (L[j]..j), (i..R[i])
//PRE: CanonicalGenerator has been calculated
private void build4nBoundList() {
    int[] RBounds; // RBounds[i] = How many times i is a right bound
    int[] LBounds; // LBounds[i] = How many times i is a left bound

    // Int arrays are initialized to 0....
    int size = Rcanonical.length - 1;
    RBounds = new int[size];
    LBounds = new int[size];

    // counting bounds for each i
    for (int i = 1; i < size; i++) {
        RBounds[i]++; // Each position is a right bound at least once
        LBounds[i]++; // Each position is a left bound at least once
        RBounds[Rcanonical[i]]++;
        LBounds[Lcanonical[i]]++;
    }

    // generate an array of bounds, ordered by increasing bound value
    size = (4 * n) + 1; // +1 needed since index 0 wont be used
    int currentIndex = 1;
    boundArray = new Bound[size];
    for (int i = 1; i < RBounds.length; i++) {
        // get all bounds and for the same i bound value, get the left ones first
        for (int k = 1; k <= LBounds[i]; k++) {
            boundArray[currentIndex] = new Bound(i, true);
            currentIndex++;
        }

        for (int k = 1; k <= RBounds[i]; k++) {
            boundArray[currentIndex] = new Bound(i, false);
            currentIndex++;
        }

    }
}
```

```
///////////////
// Utilities //
///////////////


    //Computes the reversed perm p
    // PRE: p is an unsigned permutation
    public int[] inverse() {
        int[] pInversed = new int[p.length];
        for (int i = 1; i <= n; i++)
            pInversed[p[i]] = i;

        return pInversed;
    }

    // Computes the reverse of this widened perm
    // PRE: perm is a widened unsigned permutation
    public int[] inverse(int[] perm) {
        int[] permInversed = new int[perm.length];
        for (int i = 0; i < perm.length - 1; i++)
            permInversed[perm[i]] = i;
        permInversed[perm.length - 1] = 0;

        return permInversed;
    }

    public void showStack(Stack S) {
        System.out.print("Stack: [");
        for (int i = S.size() - 1; i >= 0; i--)
            System.out.print(S.elementAt(i) + " ");
        System.out.println("]");
    }

    //hides first and last elem
    public void showNarrowIntArray(String title, int[] intArray) {
        System.out.print(title + "[");
        for (int i = 1; i < intArray.length - 1; i++) {
            System.out.print(intArray[i] + " ");
        }
        System.out.println("]");
    }



    public void showIntArray(String title, int[] intArray, int startIndex, int endIndex)
    {
        System.out.print(title + "[");
        for (int i = startIndex; i <= endIndex; i++) {
            System.out.print(intArray[i] + " ");
        }
        System.out.println("]");
    }


}
```

```
/*
 *
 * <p>Copyright: Copyright (c) 2006</p>
 *
 * <p>Company: UQAM</p>
 *
 * @author andre Levasseur
 * @version 1.0
 */


public class Bound {

     private int value;
     private boolean isLeftBound;



    //constructor
    public Bound( int value, boolean isLeftBound) {
        this.value = value;
        this.isLeftBound = isLeftBound;
    }



    //accessors
    int getValue() {
       return value;
   }

    boolean isLeftBound() {
       return isLeftBound;
   }



   //utility
   public String toString() {
       if (isLeftBound)
           return value + ":L";
       else
           return value + ":R";
   }



}
```