Algorithmique pour le consensus d'ordres

Mémoire de fin d'étude

Master *Sciences et Technologies*,
Mention *Informatique*,
Parcours Théorique

Auteur
Lisa de Mattéo

Superviseurs
Sèverine Bérard
Vincent Ranwez

Lieu de stage
ISE-M UMR5554 - CNRS, Université de Montpellier

soutenu publiquement le lundi 26 juin 2017

Avant-propos

Environ 60% du chapitre 1 d'introduction ainsi que 30% du chapitre 2 sur les relations d'ordres sont tirées de l'étude bibliographique préliminaire associée à ce mémoire, comme préconisé dans les consignes de rédaction.

Table des matières

Liste des Algorithmes

1

Introduction

L'ensemble des chromosomes présents dans les cellules d'un organisme composent son génome, c'est-à-dire son patrimoine génétique. Un chromosome contient et ordonne un ensemble de marqueurs moléculaires (*e.g.*, gènes). Avoir accès au génome des organismes est un enjeux crucial de la génomique : cela permet d'en comprendre l'organisation, les différentes transformations qu'il a connues, mais aussi de pouvoir le comparer à d'autres. Il est pour cela possible d'utiliser des techniques de séquençage, qui fragmentent le génome en de petits morceaux qu'il faut ensuite ré-assembler. Il a été montré que cette phase d'assemblage est un problème **NP**-difficile : dès lors, les heuristiques utilisées fournissent un accès imparfait au génome. Par conséquent, l'ordre entre les marqueurs fournit par l'assemblage est *partiel*, et peut même contenir des erreurs.

Une manière visuelle de représenter schématiquement ce que l'on sait de l'organisation des marqueurs moléculaires est de regrouper ensemble ceux que l'on sait être sur un même chromosome. Un partitionnement en classes d'équivalences, appelées groupes de liaison dans ce contexte biologique, est ainsi obtenu. Il est ensuite possible de représenter chacun de ces groupes de liaison par un segment, sur lequel la position (relative ou absolue) de chaque marqueur est indiquée par un tiret : il s'agit d'une *carte génomique*. Un exemple d'une carte simplifiée est donné dans la figure 1.1 à gauche. Idéalement, une carte est composée d'autant de segments qu'il y a de chromosomes, et contient autant de tirets que de marqueurs. Cependant, l'information dont on dispose est généralement incomplète : des cartes parcellaires contenant plus de segments que de chromosomes (on ne dispose pas de l'information qui permettrait de les regrouper), avec seulement un sous-ensemble des marqueurs, sont obtenues (pour les autres marqueurs, aucune information n'est disponible).

Par ailleurs, une carte génomique peut être obtenue *via* différents types de données biologiques et différentes approches méthodologiques, comme par exemple des techniques de cartographie, de déséquilibre de liaison, ou encore par des logiciels de prédiction. Par conséquent, pour une espèce donnée, de nombreuses cartes peuvent avoir été obtenues par différentes équipes de recherche, en utilisant différentes approches, ainsi qu'avec différents types de marqueurs biologiques. Des travaux récents [13, 6, 7, 11] montrent qu'il est possible, en confrontant ces différents ordonnancements, d'en proposer une synthèse plus riche et plus fiable que ce qui est obtenu par une unique approche. En effet, si les cartes ne couvrent pas les mêmes marqueurs, les combiner peut permettre de couvrir une plus grande partie du génome. De plus, des informations concordantes entre

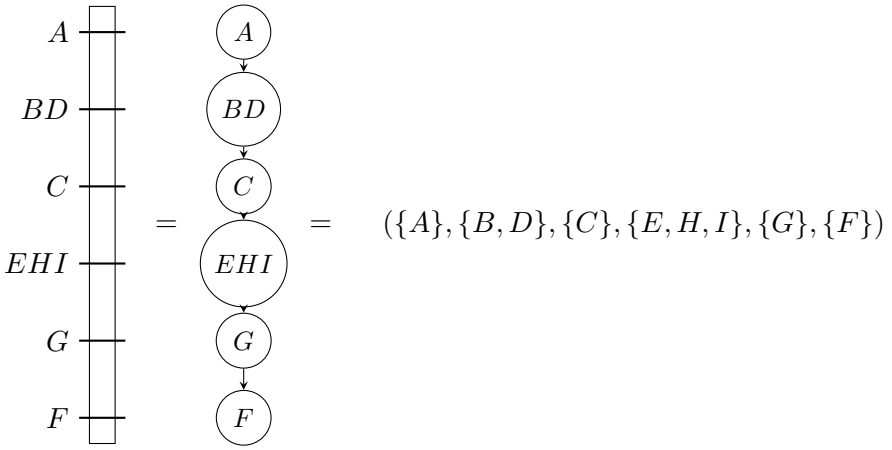


FIGURE 1.1 – Carte génomique simplifiée à gauche et l'ordre à *buckets* correspondant à droite

plusieurs cartes se verront accorder plus de crédit. Ce stage s'intéresse à la combinaison de cartes génomiques existantes, contenant des informations partielles et/ou contradictoires, en une unique, appelée *carte consensus*. Cette dernière doit représenter l'ordre le plus plausible entre les marqueurs.

Dans la réalité, un chromosome linéaire contenant un ensemble de marqueurs peut être représenté par un *ordre total* sur ces mêmes marqueurs. Cependant, comme mentionné ci-dessus, différentes imprécisions peuvent être contenues dans une carte génomique. Il arrive fréquemment que la position relative entre plusieurs marqueurs ne soit pas connue lors de la création d'une carte génomique (comme les marqueurs E, H et I sur la figure 1.1). Dès lors, ces marqueurs sont positionnés au même endroit de la carte (*i.e.*, sur le même tiret). Une telle carte peut être représentée par un *ordre à buckets*. Ce mémoire présente trois variantes du consensus de cartes génomiques (deux dans le chapitre 4 et une autre en annexe B), qui s'intéressent à proposer un consensus *sans conflit* de deux cartes génomiques. L'intérêt d'un tel consensus est d'afficher un "squelette" de l'ordre des marqueurs sur un chromosome, qui ne conserve que les informations les plus fiables.

Les différents concepts que nous utilisons sur les relations d'ordres sont introduits dans le 2e chapitre. Dans le but de simplifier la résolution des trois sous-problèmes de consensus de cartes génomiques, nous proposons dans la section 3 un pré-traitement à effectuer sur les deux ordres donnés. Les deux premiers consensus sans conflit présentés dans ce mémoire (chapitre 4) concernent les problèmes de la *plus longue sous-séquence commune* et de la *plus longue sous-séquence commune induite* à deux ordres à *buckets*, que nous définissons dans la section 4.1. Ces deux variantes étant proches, nous seront amenés à désigner les deux en même temps par la locution "plus longue sous séquence commune (induite)" ou "PLCS(I)". Les sections 4.2 et 4.3 mettent en avant deux manières de résoudre les problèmes de la plus longue sous-séquence commune (induite) à deux ordres à *buckets*.

2

Relations d'ordres

Dans ce chapitre sont données les définitions des différents types d'ensembles ordonnés étudiés lors de ce stage. Ces notions sont utilisées tout au long de ce mémoire.

Ce chapitre est en grande partie réalisée à partir des informations contenues dans [3], le reste étant des définitions usuelles.

Définition 1 (Relation binaire)**.** *Une* relation binaire R *sur un ensemble fini non vide* $\mathcal{D}$ *(appelé* domaine*) est un sous-ensemble de* $\mathcal{D} \times \mathcal{D}$ *; on note* $(x, y) \in R$ *par* $x \prec_R y$*. En outre, le symbole* $\not\asymp$ *désigne une notion d'incomparabilité :* $x \not\asymp_R y$ *signifie* $(x, y) \notin R$ *et* $(y, x) \notin R$*, i.e.* x *et* y *sont incomparables.*

Définition 2 (Ordre partiel)**.** *Une relation binaire* σ *est un* ordre partiel (strict)[1] *sur* $\mathcal{D}$ *si, pour tout* $x, y, z \in \mathcal{D}$*, elle est :*

- antiréflexive*, i.e.,* $x \not\prec_\sigma x$*,*
- antisymétrique*, i.e.,* $x \prec_\sigma y \Rightarrow y \not\prec_\sigma x$*,*
- transitive*, i.e.,* $(x \prec_\sigma y$ *et* $y \prec_\sigma z) \Rightarrow x \prec_\sigma z$*.*

Définition 3 (Ordre à *buckets*)**.** *Un ordre partiel* π *sur* $\mathcal{D}$ *est un* ordre à *buckets s'il est* négativement transitif*, c'est-à-dire que pour tout* $x, y, z \in \mathcal{D}$*,* $(x \not\prec_\pi z$ *et* $z \not\prec_\pi y) \Rightarrow x \not\prec_\pi y$*. Par conséquent, le domaine* $\mathcal{D}$ *est partitionné en une suite de buckets* $(\mathcal{B}_1, \ldots, \mathcal{B}_t)$ *telle que* $x \prec_\pi y$ *si, et seulement si, il existe* i *et* j *avec* $i < j$ *(avec* $<$ *désignant l'ordre sur les entiers naturels) et* $x \in \mathcal{B}_i$ *et* $y \in \mathcal{B}_j$*. Ainsi, dans un tel ordre, deux éléments sont incomparables si, et seulement si, ils sont dans un même bucket.*

Remarque 1. *Notons la différence entre* $e \not\asymp_\pi e'$ *(i.e.,* e *et* e' *sont dans le même bucket de* π*) et* $e \not\prec_\pi e'$ *(i.e., soit* e *et* e' *sont dans le même bucket de* π*, soit* $e' \prec_\pi e$*).*

[1] Dans la plupart des communautés, un ordre partiel est une relation réflexive alors qu'un ordre partiel strict est une relation antiréflexive ; de la même façon que dans [3, 4], un abus de langage est fait dans ce mémoire, où un ordre partiel désigne une relation antiréflexive.

Dans ce mémoire, nous représentons un *bucket* par une suite d'élements de $\mathcal{D}$. Néanmoins, deux *buckets* présentant différemment exactement le même ensemble d'éléments représentent la même information d'ordre (puisque les éléments au sein d'un même *bucket* sont incomparables). En fait, la représentation des *buckets* peut être un enjeu important concernant la complexité en temps de certains algorithmes.

Définition 4 (Ordre total)**.** *Un ordre partiel τ sur $\mathcal{D}$ est un* ordre total[2] *s'il est* complet*, i.e., pour tout $x, y \in \mathcal{D}$ avec $x \neq y$, $x \prec_\tau y$ ou $y \prec_\tau x$. Autrement dit, tous les éléments de τ sont comparables ; τ représente une permutation des éléments de $\mathcal{D}$. Un ensemble totalement ordonné désigne un ordre total.*

Finalement, tous les ordres considérés dans ce mémoire sont stricts, *i.e.*, désignent une relation antiréflexive.

Exemple 1. *Sur le domaine $\mathcal{D} = \{A, B, C, D\}$, $B \prec_\tau D \prec_\tau A \prec_\tau C$ représente un ordre total, $\{A, B\} \prec_\pi \{C, D\}$ un ordre à buckets (mais pas un ordre total), et $\{A \prec_\sigma C,\ B \prec_\sigma D\}$ un ordre partiel.*

Dans la suite de ce mémoire, nous représentons l'ordre à buckets π par $(\{A, B\}, \{C, D\})$.

On peut remarquer qu'un ordre total est un ordre à *buckets* (avec un seul élément par *bucket*), et qu'un ordre à *buckets* est un ordre partiel (tout ordre est partiel).

Définition 5 (Ordre lexicographique sur un produit cartésien)**.** *Soient A et B deux ensembles ordonnés. L'*ordre lexicographique *sur le produit cartésien $A \times B$ est défini par $(a, b) < (a', b') \Leftrightarrow (a < a')$ ou $(a = a'$ et $b < b')$, $\forall a, a' \in A$ et $\forall b, b' \in B$. En outre, si A et B sont totalement ordonnés, alors l'ordre lexicographique sur $A \times B$ l'est aussi.*

Définition 6 (Extension linéaire d'un ordre)**.** *Par ailleurs, une* extension linéaire *d'un ordre σ sur un domaine $\mathcal{D}$ est un ordre total τ sur $\mathcal{D}$ qui ne contredit pas σ, i.e., $\forall e_1, e_2 \in \mathcal{D}$, $e_1 \prec_\sigma e_2 \Rightarrow e_1 \prec_\tau e_2$. $L(\sigma)$ désigne l'ensemble des extensions linéaires d'un ordre σ.*

Exemple 2. *Soit $\pi = (\{A, B\}, \{C, D\})$ un ordre à buckets sur le domaine $\mathcal{D} = \{A, B, C, D\}$. Les ordres totaux $A \prec_\tau B \prec_\tau C \prec_\tau D$, $A \prec_\tau B \prec_\tau D \prec_\tau C$, $B \prec_\tau A \prec_\tau C \prec_\tau D$ et $B \prec_\tau A \prec_\tau D \prec_\tau C$ sont les seules extensions linéaires de π.*

Définition 7 (Sous-séquence d'un ordre total)**.** *Soit τ un ordre total sur un domaine $\mathcal{D}$. Une* sous-séquence *de τ est une suite $s = (e_1, e_2, \ldots, e_l)$ d'éléments de $\mathcal{D}$ telle que, pour tout e_i et e_{i+1} deux éléments consécutifs de s, $1 \leq i \leq l - 1$, $e_i \prec_\tau e_{i+1}$.*

[2] Dans la plupart des communautés, un ordre est *total* si tous ses éléments sont comparables, et il est *partiel* sinon (autrement dit, un ordre ne peut pas être total et partiel à la fois). Dans ce mémoire et de la même façon que dans [3, 4], un ordre total désigne une généralisation d'un ordre partiel.

Définition 8 (Sous-séquence commune et PLSC à deux ordres totaux)**.** *Soient τ_1 et τ_2 deux ordres totaux de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$. Une sous-séquence* commune *à τ_1 et τ_2 est une sous-séquence à la fois de τ_1 et de τ_2. La* taille *(ou* longueur*) d'une sous-séquence commune est définie par le nombre d'éléments de $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$ qu'elle contient. Une sous-séquence commune de taille maximum est appelée plus longue sous-séquence commune (PLSC).*

Exemple 3. *Soient $\tau_1 = \{A \prec_\tau B \prec_\tau D \prec_\tau C\}$ et $\tau_2 = \{B \prec_\tau A \prec_\tau C \prec_\tau D\}$ deux ordres totaux d'un même domaine $\mathcal{D} = \{A, B, C, D\}$.*

Les suites (A), (A, D), (A, B, D, C) et (A, C) sont des sous-séquences de τ_1, et les suites (B, C, D), (C, D), (B, A) et (B, A, C, D) des sous-séquences de τ_2.

Les suites (A), (D), (A, C), (A, D), (B, D) et (B, C) sont des sous-séquences communes à τ_1 et τ_2. La suite (A, D) est une PLSC à τ_1 et τ_2.

Définition 9 (Sous-séquence augmentante et PLSA d'un ordre total)**.** *Soit τ un ordre total sur $\mathcal{D} = \{1, \ldots, |\mathcal{D}|\}$. Une sous-séquence augmentante de τ est une suite $s = (e_1, e_2, \ldots, e_l)$ d'éléments de $\mathcal{D}$ telle que pour tout e_i et e_{i+1} deux éléments consécutifs de s, $1 \leq i \leq l-1$, $e_i \prec_\tau e_{i+1}$ et $e_i < e_{i+1}$ (avec $<$ l'ordre sur les entiers naturels). La taille de s est l, i.e., son nombre d'éléments ; une sous-séquence augmentante de taille maximum est appelée une plus longue sous-séquence augmentante (PLSA).*

Exemple 4. *Soient $5 \prec_\tau 2 \prec_\tau 1 \prec_\tau 3 \prec_\tau 4$ un ordre total τ sur le domaine $\mathcal{D} = \{1, 2, 3, 4, 5\}$.*

Les suites $(2, 3, 4)$ et $(1, 3, 4)$ sont les deux seules PLSA de τ.

Maintenant que nous avons introduit les définitions usuelles sur les relations d'ordres utilisées dans ce rapport, nous donnons dans le chapitre suivant un pré-traitement pour le consensus de deux ordres à *buckets*.

3

Un pré-traitement d'homogénéisation

Dans ce chapitre, nous proposons un pré-traitement pour le consensus de deux ordres à *buckets* π_1 et π_2 sur les domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ respectivement. Ce procédé, décrit par l'algorithme 1, permet de modifier π_1 et π_2 en deux ordres à *buckets* π_1^h et π_2^h de façon à faciliter l'obtention d'une sous-séquence commune (induite) à π_1 et π_2. Nous appelons cette méthode une "homogénéisation".

La section 3.1 donne la définition d'une homogénéisation ainsi qu'une première propriété sur les ordres homogénéisés, ce qui permet par la suite à la section 3.2 de donner un premier algorithme permettant d'homogénéiser π_2 par rapport à π_1 en $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$ (avec, sans perte de généralité, $|\mathcal{D}_1| \leq |\mathcal{D}_2|$). Par ailleurs, en affinant la bibliographie, nous avons pris connaissance d'un article [2] dans lequel les auteurs définissent, sans donner d'algorithme ni de complexité, un *raffinement* de la même manière que nous avons défini une homogénéisation. En cherchant ce mot-clé de raffinement, nous avons découvert un second article [5] dans lequel les auteurs affirment qu'un raffinement peut être implémenté en temps linéaire, sans donner plus de détails. Nous proposons donc un second algorithme dans la section 3.3 qui permet d'atteindre cette complexité.

3.1 Définition

Dans cette section, nous donnons la définition et un exemple d'homogénéisation, ainsi qu'une première propriété sur les *buckets* des ordres homogénéisés.

Définition 10 (Homogénéisation de deux ordres à *buckets*)**.** *Soit π_1 et π_2 deux ordres à buckets de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$. Soit $\mathcal{B}^1 = (B_1^1, \ldots, B_{|\mathcal{B}^1|}^1)$ (resp. $\mathcal{B}^2 = (B_1^2, \ldots, B_{|\mathcal{B}^2|}^2)$) la suite ordonnée des buckets de π_1 (resp. π_2). Soit $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$. Sont associés à chaque élément e de $\mathcal{D}$ $pos_1(e)$ et $pos_2(e)$, deux entiers indiquant respectivement le numéro du bucket de π_1 et π_2 contenant l'élément e.*

*L'*homogénéisation *de π_1 et π_2 associe respectivement à ces deux ordres deux ordres à buckets π_1^h et π_2^h sur $\mathcal{D}$ qui sont tels que, pour tout élément $e, e' \in \mathcal{D}$:*

- *e et e' appartiennent au même bucket de π_1^h (resp. π_2^h) si, et seulement si, ils sont dans un même bucket dans π_1 **et** dans un même bucket dans π_2, i.e., si, et seulement si, $pos_1(e) = pos_1(e')$ et $pos_2(e) = pos_2(e')$,*

- *le bucket de π_1^h (resp. π_2^h) contenant e est situé avant le bucket de π_1^h (resp. π_2^h) contenant e' si, et seulement si, $pos_1(e) < pos_1(e')$ ou $[pos_1(e) = pos_1(e')$ et $pos_2(e) < pos_2(e')]$ (resp. $pos_2(e) < pos_2(e')$ ou $[pos_2(e) = pos_2(e')$ et $pos_1(e) < pos_1(e')]$).*

On appelle π_1^h l'homogénéisation de π_1 par rapport à π_2, et π_2^h l'homogénéisation de π_2 par rapport à π_1.

Exemple 5. *Soient*
$\pi_1 = (\{k\}, \{a, b\}, \{l, c\}, \{d, e, f\}, \{i, j\}, \{g, h\})$ *et*
$\pi_2 = (\{g, \}, \{c, d, e, f\}, \{m, q\}, \{r, a\}, \{b, n\}, \{o, p\})$ *deux ordres à buckets de domaines respectifs* $\mathcal{D}_1 = \{a, b, c, d, e, f, g, h, i, j, k, l\}$ *et* $\mathcal{D}_2 = \{a, b, c, d, e, f, g, h, m, n, o, p, q, r\}$.
L'ordre $\pi_1^h = (\{a\}, \{b\}, \{c\}, \{d, e, f\}, \{g, h\})$ *est l'homogénéisation de* π_1 *par rapport à* π_2.
L'ordre $\pi_2^h = (\{g, h\}, \{c\}, \{d, e, f\}, \{a\}, \{b\})$ *est l'homogénéisation de* π_2 *par rapport à* π_1.

Propriété 1. *Soient π_1 et π_2 deux ordres à buckets de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$, et π_1^h et π_2^h les ordres à buckets respectifs issus de leurs homogénéisations. π_1^h et π_2^h contiennent les mêmes buckets mais pas forcément dans le même ordre.*

Démonstration. Par l'absurde. Supposons qu'il existe e_1 et e_2 de $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$ tels qu'ils appartiennent au même *bucket* de π_1^h et à des *buckets* différents de π_2^h. Comme e_1 et e_2 appartiennent au même *bucket* de π_1^h, par définition, $pos_1(e_1) = pos_1(e_2)$ et $pos_2(e_1) = pos_2(e_2)$. Par conséquent, e_1 et e_2 appartiennent au même *bucket* de π_2^h : il s'agit d'une contradiction. De la même façon, il est impossible que deux éléments appartiennent au même *bucket* de π_2^h et à des *buckets* différents de π_1^h. □

Remarque 2. *Par définition, les ordres homogénéisés contiennent exactement les éléments de $\mathcal{D}_1 \cap \mathcal{D}_2$.*

Maintenant que nous avons défini l'homogénéisation, nous donnons dans la section suivante un algorithme en $\mathcal{O}(|\mathcal{D}_2| log(|\mathcal{D}_1|))$ (avec, sans perte de généralité, $|\mathcal{D}_1| \leq |\mathcal{D}_2|$) qui, étant donné deux ordres à *buckets* π_1 et π_2 sur $\mathcal{D}_1$ et $\mathcal{D}_2$ respectivement, retourne π_2^h, l'ordre homogénéisé de π_2 par rapport à π_1.

3.2 Algorithme 1 : Homogénéisation

Avant de présenter l'algorithme 1 d'homogénéisation, nous montrons un premier lemme, qui décrit une façon simple d'homogénéiser un ordre à *buckets* π_2 par rapport à un autre ordre à *buckets* π_1, lorsque les éléments des *buckets* de π_2 sont ordonnés selon leurs numéros de *bucket* dans π_1.

Lemme 1. *Soient π_1 et π_2 deux ordres à buckets sur $\mathcal{D}_1$ et $\mathcal{D}_2$ respectivement, ainsi que $\mathcal{B}^1 = (B_1^1, \ldots, B_{|\mathcal{B}^1|}^1)$ et $\mathcal{B}^2 = (B_1^2, \ldots, B_{|\mathcal{B}^2|}^2)$ leurs suites respectives de buckets,*

tels que chaque bucket est représenté par une suite ordonnée de ses éléments. On note $B_i'^2$ *le bucket* B_i^2 *de* π_2 *privé des éléments de* $\mathcal{D}_1 \backslash \mathcal{D}_2$*, i.e.* $B_i'^2 = B_i^2 \backslash (\mathcal{D}_1 \backslash \mathcal{D}_2)$*. Si au sein de chaque bucket de* π_2*, les éléments sont triés selon l'ordre croissant de leurs numéros de bucket dans* π_1*, alors l'homogénéisation de* π_2 *par rapport à* π_1*, notée* π_2^h*, est obtenue à partir de* π_2 *en scindant chacun de ses buckets* $B_i'^2 = (e_{i_1}, \ldots, e_{i_{|B_i'^2|}})$ *entre deux éléments consécutifs* e_{i_s} *et* $e_{i_{s+1}}$ *si, et seulement si,* e_{i_s} *et* $e_{i_{s+1}}$ *appartiennent à des buckets différents dans* π_1 *(*$\forall 1 \leq s \leq |B_i^2| - 1$*).*

Démonstration. Montrons que π_2^h est l'homogénéisation de π_2 par rapport à π_1 avec les points suivants :

- **Le domaine de** π_2^h **est** $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$**.** π_2^h est obtenu à partir de π_2 en sectionnant ses *buckets* B_i^2 privé des éléments de $\mathcal{D}_1 \backslash \mathcal{D}_2$: π_2^h est donc sur le domaine $\mathcal{D}$.

- **Montrons que deux éléments** e **et** e' **de** $\mathcal{D}$ **appartiennent au même *bucket* dans** π_2^h **si, et seulement si, ils sont dans le même *bucket* dans** π_1 **ainsi que dans** π_2**.**

  $\Leftarrow$ Par l'absurde. Supposons que e et e' soient dans des *buckets* différents de π_2^h. Du coup, ils sont dans un *bucket* de π_2 qui a été sectionné quelque part entre e et e'. Cependant, un *bucket* est sectionné seulement entre des éléments qui n'appartiennent pas au même *bucket* dans π_1. Comme les *buckets* de π_2 sont ordonnés selon l'ordre croissant des numéros de *bucket* dans π_1, tous les éléments entre e et e' (eux compris) appartiennent au même *bucket* de π_1. Il s'agit d'une contradiction.

  $\Rightarrow$ Si e et e' sont dans le même *bucket* de π_2^h, c'est qu'ils sont dans le même *bucket* dans π_2 (la construction ne fait que scinder des *buckets* de π_2). Reste à montrer que ces deux éléments sont dans le même *bucket* dans π_1. Par l'absurde, supposons qu'ils appartiennent à des *buckets* différents dans π_1. Pour se retrouver dans le même *bucket* de π_2^h, c'est que le *bucket* de π_2 auquel ils appartiennent n'a été scindé à aucun endroit entre e et e'. Autrement dit, dans ce *bucket*, tous les éléments entre e et e' appartiennent au même *bucket* de π_1 : il s'agit d'une contradiction.

- **Montrons que** $\forall e, e' \in \mathcal{D}$**,** $e \prec_{\pi_2^h} e' \Leftrightarrow e \prec_{\pi_2} e'$ **ou (**$e \not\asymp_{\pi_2} e'$ **et** $e \prec_{\pi_1} e'$**)**

  $\Leftarrow$ Si e est dans un *bucket* qui est avant le *bucket* contenant e' dans π_2 (*i.e.*, $e \prec_{\pi_2} e'$), alors par construction, le *bucket* de π_2^h contenant l'élément e de $\mathcal{D}$ est situé avant le *bucket* de π_2^h contenant l'élément e' (*i.e.*, $e \prec_{\pi_2^h} e'$).

  Si e et e' sont dans le même *bucket* B de π_2 mais que e est dans un *bucket* situé avant le *bucket* contenant e' dans π_1 (*i.e.*, $e \not\asymp_{\pi_2} e'$ et $e \prec_{\pi_1} e'$), alors e est avant e' dans le *bucket* B de π_2 (puisque les *buckets* de π_2 sont triés selon l'ordre croissant des numéros de *bucket* dans π_1). Soit e'' le premier élément suivant e dans le *bucket* B de π_2 qui est tel que e et e'' ne sont pas dans le même *bucket* de π_1 (un tel élément existe toujours ; il peut s'agir de e'). Le *bucket* B est alors scindé entre e'' et son

prédécesseur, et e se retrouve dans un *bucket* situé avant le *bucket* contenant e' dans π_2^h.

$\Rightarrow$ Si e est dans un *bucket* situé avant celui contenant e' dans π_2^h (*i.e.*, $e \prec_{\pi_2^h} e'$), c'est soit qu'ils sont dans des *buckets* différents de π_2 (avec celui contenant e situé avant celui contenant e'), soit qu'ils sont dans un même *bucket* B de π_2 (avec e apparaissant avant e' dans B) qui été scindé entre e et e'. Pour cela, il existe dans B un élément e'' (il peut s'agir de e') entre e et e' tel que e'' n'est pas dans le même *bucket* de π_1 que son prédécesseur direct dans B. Du coup, il est impossible que e et e' soient dans un même *bucket* de π_1 (puisque les *buckets* de π_2 sont triés selon l'ordre croissant des numéros de *bucket* dans π_1).

□

L'algorithme 1 page 10 retourne l'homogénéisation π_2^h d'un ordre à *buckets* π_2 sur $\mathcal{D}_2$ par rapport à deuxième ordre à *buckets* π_1 sur $\mathcal{D}_1$, en temps $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$ (avec, sans perte de généralité, $|\mathcal{D}_1| \leq |\mathcal{D}_2|$).

Cet algorithme se compose de deux actions :

- trier les éléments des *buckets* de π_2 selon leurs numéros de *bucket* dans π_1 (lignes 2-3, 6-7 et 8)
- scinder les *buckets* de π_2 pour effectuer l'homogénéisation (lignes 9 à 18).

Proposition 1 (Terminaison de l'algorithme 1)**.** *L'algorithme 1 termine.*

Démonstration. Les boucles **"pour"** lignes 2, 3, 4, et 6 terminent puisqu'elles itèrent sur des éléments finis. La boucle **"pour"** de la ligne 11 parcourt les éléments de la liste *LtempSort* : cette dernière contient les mêmes éléments que la liste *Ltemp*, qui possède un nombre fini d'éléments. En effet, chacun de ces derniers est inséré dans *Ltemp* par la ligne 7, qui s'exécute un nombre fini de fois. □

Proposition 2 (Correction de l'algorithme 1)**.** *L'algorithme 1 retourne l'homogénéisation de π_1 et π_2, les deux ordres à buckets qui lui sont donnés en entrée.*

Démonstration. **Montrons qu'à chaque i^{e} fin d'instruction de la ligne 8, la liste *LtempSort* contient exactement les éléments du i^{e} *bucket* de π_2, triés selon l'ordre croissant de leurs numéros de *bucket* dans π_1.**

Les lignes 2 et 3 permettent de remplir le dictionnaire de hashage dic_1, qui contient, pour chaque élément e de $\mathcal{D}$, le couple (e, i) si, et seulement si, l'élément e est dans le i^{e} *bucket* de π_1.

La ligne 4 parcourt les *buckets* B_i^2 de π_2, de B_1^2 à $B_{|\mathcal{B}^2|}^2$, et pour chaque élément e de B_i^2, les lignes 6 et 7 ajoutent le couple $(e, dic_1.\text{getValue}(e))$ dans la liste *Ltemp*. Ensuite, la ligne 8 créée la liste *LtempSort*, qui contient exactement les couples de *Ltemp* triés selon leurs secondes positions, *i.e.*, selon les numéros de *bucket* dans π_1.

Algorithme 1 : HOMOGÉNÉISATION

Données : Deux ordres à *buckets* π_1 et π_2 non vides de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$.

Résultat : π_2^h, l'homogénéisation de π_2 par rapport à π_1.

Soient $\mathcal{B}^1 = (B_1^1, \ldots, B_{|\mathcal{B}_1|}^1)$ et $\mathcal{B}^2 = (B_1^2, \ldots, B_{|\mathcal{B}_2|}^2)$ les suites respectives des *buckets* de π_1 et π_2, dic_1 et dic_2 deux dictionnaires de hashage vides, ainsi que π_2^h un ordre à *buckets* vides.

pour i *de* 1 *à* $|\mathcal{B}^1|$ **faire** `// Buckets de` π_1
  pour e *de* $B_i^1[1]$ *à* $B_i^1[|B_i^1|]$ **faire** $dic_1.insert(key = e, value = i)$;

pour i *de* 1 *à* $|\mathcal{B}^2|$ **faire**
  Soit $Ltemp$ une liste vide.
  pour e *de* $B_i^2[1]$ *à* $B_i^2[|B_i^2|]$ **faire**
    si $e \in dic_1$ **alors** $Ltemp$.push(e,dic_1.getValue(e)) ;
  `// tri croisant de` $Ltemp$ `selon les 2nd positions de ses couples.`
  $LtempSort \leftarrow$ sort($Ltemp$) ;
  $prevPos \leftarrow LtempSort[1][2]$;
  Soit $Buck$ un *bucket* vide.
  pour (e, pos) *de* $LtempSort[1]$ *à* $LtempSort[|LtempSort|]$ **faire**
    si $prevPos = pos$ **alors** `// enrichit le bucket courrant`
      $Buck$.add(e) ;
    sinon `// new bucket`
      π_2^h.push_back($Buck$) ;
      $Buck \leftarrow$ new Buck(e) ;
      $prevPos \leftarrow pos$;
  `// gestion du dernier bucket`
  π_2^h.push_back($Buck$) ;

retourner π_2^h ;

Lorsque les *buckets* de π_2 sont ainsi triés, le lemme 1 garantit que l'ordre π_2^h est obtenu à partir de π_2 en scindant chacun de ses *buckets* entre deux éléments consécutifs e et e' si, et seulement si, ils appartiennent à des *buckets* différents de π_1. Montrons que c'est ce que font les lignes 9 à 18.

La boucle **"pour"** ligne 11 parcourt la liste triée $LtempSort$, de $LtempSort[1]$ à $LtempSort[|LtempSort|]$. Autrement dit, cette boucle parcourt tous les éléments e du i^{e} *bucket* de π_2, dans l'ordre de leurs numéros de *bucket* dans π_1. La ligne 13 ajoute e dans le *bucket* $Buck$ en construction si, et seulement si, e est le premier élément de $LtempSort$ à être parcouru, ou s'il est dans le même *bucket* de π_1 que son prédécesseur immédiat dans le parcourt (*i.e.*, si $prevPos = pos$). Sinon, le *bucket* $Buck$ en construction à l'itération précédente est inséré à la fin de π_2^h (ligne 15), puis $Buck$ est réinitialisé à e (ligne 16). □

Proposition 3 (Complexité en temps de l'algorithme 1)**.** *Sans perte de généralité, disons que* $|\mathcal{D}_1| \leq |\mathcal{D}_2|$. *L'algorithme 1 nécessite* $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$ *en temps.*

Démonstration. Les boucles lignes 2 et 3 sont exécutées au plus $|\mathcal{D}_1|$ fois, puisqu'elles parcourent les éléments de π_1. Comme une insertion dans un dictionnaire de hashage est au pire des cas logarithmique en sa taille, l'instruction de la ligne 3 est en $\mathcal{O}(log(|\mathcal{D}_1|))$. Du coup, les lignes 2 à 3 ont une complexité de $\mathcal{O}(|\mathcal{D}_1|log(|\mathcal{D}_1|))$.

De plus, les boucles lignes 4 et 6 parcourent les éléments de $e \in \mathcal{D}_2$. Le test réalisé à la ligne 7 peut s'implémenter en $\mathcal{O}(log(|\mathcal{D}_1|))$. La récupération d'une valeur dans le dictionnaire de hashage dic_1 prend $\mathcal{O}(log(|\mathcal{D}_1|))$, et l'ajout dans la liste $Ltemp$ peut se faire en temps $\mathcal{O}(1)$: la ligne 7 s'exécute donc totalement en $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$. L'ensemble des tris effectués à la ligne 8 s'exécutent totalement (complexité amortie) en $\mathcal{O}(\sum_{i=1}^{|\mathcal{B}^2|} |B_i'^2|log(|B_i'^2|)) \leq \mathcal{O}(\sum_{i=1}^{|\mathcal{B}^2|} |B_i'^2|log(|\mathcal{D}|)) = \mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$, avec $|B_i'^2|$ la taille du i^{e} *bucket* de π_2 privé des éléments de $\mathcal{D}_1 \backslash \mathcal{D}_2$, et $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$. La boucle 11 parcourt les éléments de la liste $LtempsSort$, qui sont au nombre de $|B_i'^2|$ à la i^{e} itération : les instructions qu'elle contient peuvent toutes s'implémenter en temps constant, sa complexité totale (amortie) est donc en $\mathcal{O}(\sum_{i=1}^{|\mathcal{B}^2|} |B_i'^2|) = \mathcal{O}(|\mathcal{D}|)$. La complexité totale de la ligne 4 est donc en $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|) + |\mathcal{D}|log(|\mathcal{D}|) + |\mathcal{D}|) = \mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$.

Finalement, l'algorithme 1 peut s'implémenter en temps $\mathcal{O}((|\mathcal{D}_1| + |\mathcal{D}_2|)log(|\mathcal{D}_1|))$. □

Remarque 3. *La complexité de cet algorithme peut se décomposer en la complexité de sa première action : trier les buckets de* π_2 *en* $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$*, plus celle de sa deuxième action : l'homogénéisation proprement dite en* $\mathcal{O}(|\mathcal{D}|)$*. Pour cela, il suffit de sortir les lignes 5 à 8 de la boucle 4, e.g. en exécutant ces instructions avant ladite boucle, et en stockant dans* $|\mathcal{B}^2|$ *tableaux les* $|\mathcal{B}^2|$ *listes LtempSort. Cela permet de réduire la complexité de la boucle 4 de* $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|))$ *à* $\mathcal{O}(|\mathcal{D}|)$*.*

Comme mentionné dans l'introduction de ce chapitre, en affinant la bibliographie, nous nous sommes aperçus que certains auteurs affirment qu'il est possible de réaliser une homogénéisation en temps linéaire. Dès lors, en adaptant les deux premières étapes d'un

pré-traitement de [1], nous proposons, dans la section suivante, un algorithme permettant d'effectuer le tri des *buckets* de π_2 en temps linéaire.

3.3 Amélioration de la complexité

L'algorithme 2 page 13, qui trie les éléments des *buckets* de π_2 en fonction de leurs numéros de *bucket* dans π_1 est inspiré de [1]. Dans cet article, les deux ordres sont sur un même domaine ; nous l'ajustons pour des ordres ayant des domaines différents.

Proposition 4 (Terminaison de l'algorithme 2)**.** *L'algorithme 2 termine.*

Démonstration. Toutes les boucles **"pour"** itèrent sur des éléments finis. □

Proposition 5 (Correction de l'algorithme 2)**.** *L'algorithme 2 retourne π_1 sur $\mathcal{D}_1$ et π_2^h restreint à $\mathcal{D}_1 \cap \mathcal{D}_2$, tels que les éléments des buckets de π_2 sont triés selon leurs numéros de bucket dans π_1.*

Démonstration. Nous démontrons cette proposition avec les deux propriétés suivantes.

Propriété 2. *Après la dernière itération de la boucle **"pour"** ligne 14, les éléments présents dans π_2 sont ceux qui sont présents dans π_1 et dans π_2 au début de l'algorithme.*

△ Le domaine de π_1 est renommé par les lignes 3 à 6 en $\{1, \ldots, |\mathcal{D}_1|\}$.

Chacun des *buckets* de π_2 sont vidés par la ligne 13. Ensuite, la ligne 17 ajoute l'élément $B_i^1[j]$ dans un *bucket* de π_2 si, et seulement si, $buc_2[*(B_i^1[j])] \neq -1$. Or, le tableau buc_2 contient -1 à la case $*(B_i^1[j])$ si, et seulement si, le $*(B_i^1[j])^{\text{e}}$ élément de π_1 n'appartient pas à π_2.

Finalement, pour chaque élément $B_i^1[j]$ de π_1, la ligne 17 ajoute $B_i^1[j]$ dans le $buc_2[*(B_i^1[j])]^{\text{e}}$ *bucket* de π_2 si, et seulement si, il appartient à π_2. △

Propriété 3. *Après la dernière itération de la boucle **"pour"** ligne 14, les éléments des buckets de π_2 sont triés selon leurs numéros de bucket dans π_1.*

△ Les boucles lignes 9 et 10 parcourent les éléments selon leur ordre d'apparition dans π_2, pour remplir le tableau buc_2, en associant $indBuc_2$ à la case $*(B_i^2[j])$ si, et seulement si, l'élément $B_i^2[j]$ (ayant pour valeur $*(B_i^2[j])$) est dans le $indBuc_2^{\text{e}}$ *bucket* de π_2. La boucle ligne 13 vide ensuite chaque *bucket* de π_2. Les boucles lignes 14 et 15 parcourent les éléments $B_i^1[j]$ de π_1, et pour chacun d'entre eux, ajoute un pointeur vers $B_i^1[j]$ à la fin du $buc_2[*(B_i^1[j])]^{\text{e}}$ *bucket* de π_2 si $B_i^1[j]$ est présent dans π_2 au début de l'algorithme. Du coup, les éléments ont bien été insérés, dans chaque *bucket* de π_2, selon leurs numéros de *bucket* dans π_1, puisque, suite au renommage, ils apparaissent dans l'ordre croissant de leurs valeurs dans π_1. △

□

Proposition 6 (Complexité en temps de l'algorithme 2)**.** *L'algorithme 2 nécessite $\mathcal{O}(|\mathcal{D}|)$ en temps.*

Algorithme 2 : RENOMM'TRI

Données : Deux ordres à *buckets* π_1 et π_2 non vides de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$, tels que leurs *buckets* sont représentés par des suites ordonnées, un élément e est de type pointeur vers sa valeur $*(e)$ (l'élément e de π_1 pointe vers le même espace mémoire que l'élément e de π_2), et aucune des valeurs $*(e)$ n'appartient à $\{1, \ldots, |\mathcal{D}_1|\}$.

Résultat : π_1 et π_2, tels que les valeurs pointées par les éléments de π_1 sont renommées en $\{1, \ldots, |\mathcal{D}_1|\}$ dans l'ordre d'apparition des éléments dans π_1 (même au sein d'un *bucket* puisqu'ils sont représentés par des suites ordonnées), et les éléments des *buckets* de π_2 sont triés selon leur ordre d'apparition dans π_1.

Soient $\mathcal{B}^1 = (B^1_1, \ldots, B^1_{|\mathcal{B}_1|})$ et $\mathcal{B}^2 = (B^2_1, \ldots, B^2_{|\mathcal{B}_2|})$ les suites respectives des *buckets* de π_1 et π_2, buc_2 un tableau vide, ainsi que π^h_1 et π^h_2 deux ordres à *buckets* vides.

`// Renommage du domaine`
$cpt \leftarrow 1$;
pour i *de* 1 *à* $|\mathcal{B}^1|$ **faire**
  pour j *de* 1 *à* $|B^1_i|$ **faire**
    $*(B^1_i) \leftarrow cpt$;
    $cpt++$;

`// Tri de chaque bucket de` π_2 `selon l'ordre croissant des éléments`
pour i *de* 1 *à* $|\mathcal{D}_1|$ **faire** $buc_2[i] \leftarrow -1$;
$indBuc_2 \leftarrow 1$;
pour i *de* 1 *à* $|\mathcal{B}^2|$ **faire**
  pour j *de* 1 *à* $|B^2_i|$ **faire**
    si $1 \leq *(B^2_i[j]) \leq |\mathcal{D}_1|$ **alors** $buc_2[*(B^2_i[j])] \leftarrow indBuc_2$;// $B^2_i \in \mathcal{D}_1 \cap \mathcal{D}_2$
  $indBuc_2++$;
pour i *de* 1 *à* $|\mathcal{B}^2|$ **faire** $B^2_i \leftarrow \emptyset$;
pour i *de* 1 *à* $|\mathcal{B}^1|$ **faire**
  pour j *de* 1 *à* $|B^1_i|$ **faire**
    si $buc_2[*(B^1_i[j])] \neq -1$ **alors**
      $B^2_{buc_2[*(B^1_i[j])]}.push\_back(B^1_i[j])$; // $B^1_i \in \mathcal{D}_1 \cap \mathcal{D}_2$

retourner π_1 *et* π_2 ;

Démonstration. Les boucles lignes 3 et 4 sont en $\mathcal{O}(|\mathcal{D}|)$. Les boucles lignes 9 et 10 aussi. La boucle ligne 13 est en $\mathcal{O}(|\mathcal{B}^2|)$, avec $|\mathcal{B}^2| \leq \mathcal{D}$. Les boucles lignes 14 et 15 sont en $\mathcal{O}(|\mathcal{D}|)$. Les autres instructions sont en $\mathcal{O}(1)$. □

Maintenant que nous avons présenté un pré-traitement permettant de faciliter le consensus de deux ordres à *buckets*, nous définissons (section 4.1) et résolvons de deux manières (sections 4.2 et 4.3) le problème de la plus longue sous-séquence commune (induite) à deux ordres à *buckets* dans le chapitre suivant.

4

Plus longue sous-séquence commune (induite) à deux ordres à *buckets*

Dans ce chapitre, nous nous intéressons au problème cherchant une plus longue sous-séquence commune (induite) (PLSC(I)) à deux ordres à *buckets*. Tout d'abord, nous donnons la définition d'une PLSC et d'une PLSCI à deux ordres à *buckets* dans la section 4.1. Dans un second temps, nous proposons dans la section 4.2 une première résolution du problème, qui adapte un algorithme de programmation dynamique connu de la littérature [12]. Dans un dernier temps, nous proposons dans la section 4.3 une autre manière de résoudre ce problème de PLSC(I) à deux ordres à *buckets*, qui passe par le calcul d'une PLSA.

4.1 Définition et propriétés d'une plus longue sous-séquence commune (induite) à deux ordres à *buckets*

Une PLSC est usuellement définie sur des ordres totaux (voir chapitre 2). Dans cette section, nous proposons une définition d'une PLSC à deux ordres à *buckets*. Intuitivement, une PLSC à deux ordres à *buckets* peut être définie de deux façons, ce qui donne lieu aux définitions 12 et 13, que nous avons nommées plus longue sous-séquence commune (PLSC) et plus longue sous-séquence commune induite (PLSCI). De plus, nous montrons un ensemble de propriétés sur les PLSC(I) à deux ordres à *buckets*, qui permettent finalement de prouver les corollaires 1 et 2. Ces derniers garantissent que l'ensemble des PLSC(I) à π_1 et π_2 est égal à celui des PLSC(I) à π_1^h et π_2^h (les homogénéisations respectives de π_1 et π_2). Cela nous permet de travailler sur les ordres homogénéisés sans perte de solutions.

Avant de définir une sous-séquence commune (induite) à deux ordres à *buckets*, nous donnons la définition d'une sous-séquence d'un ordre à *buckets*.

Définition 11 (Sous-séquence d'un ordre à *buckets*)**.** *Soit π un ordre à buckets sur un domaine $\mathcal{D}$. Une* sous-séquence *de π est une suite $s = (e_1, e_2, \ldots, e_l)$ d'éléments de $\mathcal{D}$ qui telle que, pour tout e_i et e_{i+1} deux éléments consécutifs de s, $1 \leq i \leq l-1$, soit*

$e_i \not\preceq_\pi e_{i+1}$ (e_i et e_{i+1} sont dans un même bucket), soit $e_i \prec_\pi e_{i+1}$ (le bucket contenant e_i est situé avant celui contenant e_{i+1}).

Exemple 6. *Soit $\pi_1 = (\{k\}, \{a, b\}, \{l, c\}, \{d, e, f\}, \{i, j\}, \{g, h\})$ un ordre à buckets de domaine $\mathcal{D}_1 = \{a, b, c, d, e, f, g, h, i, j, k, l\}$.*

Les suites $(k, b, a, c, l, f, e, d, j, i, g, h)$, (k, a, b, c, d, e, h), (b, a, c, i, h) et (l) sont des sous-séquences de π_1 (il en existe d'autres).

Remarque 4. *Une extension linéaire d'un ordre à buckets sur $\mathcal{D}$ correspond à une sous-séquence de cet ordre qui contient tous les éléments de $\mathcal{D}$, e.g. la sous-séquence $(k, b, a, c, l, f, e, d, j, i, g, h)$ de l'exemple précédent.*

Nous pouvons maintenant définir la notion de sous-séquence commune à deux ordres à *buckets*.

Définition 12 (Sous-séquence commune et PLSC à deux ordres à *buckets*)**.** *Soient π_1 et π_2 deux ordres à buckets de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$. Une sous-séquence commune s à π_1 et π_2 est une sous-séquence à la fois de π_1 et de π_2. La taille de s est son nombre d'éléments ; une sous-séquence commune de taille maximum est appelée une plus longue sous-séquence commune (PLSC).*

Exemple 7. *Soient π_1 de l'exemple précédent et $\pi_2 = (\{g, h\}, \{c, d, e, f\}, \{m, q\}, \{r, a\}, \{b, n\}, \{o, p\})$ un ordre à buckets sur $\mathcal{D}_2 = \{a, b, c, d, e, f, g, h, m, n, o, p, q, r\}$.*

Les suites (a, b), (g, h), (h, g), (c, e), (c, d, e, f), (c, e, f, d) sont des sous-séquences communes à π_1 et π_2 (il en existe d'autres). Il existe six PLSC à π_1 et π_2 : (c, d, e, f), (c, d, f, e), (c, e, d, f), (c, e, f, d), (c, f, d, e), et (c, f, e, d), qui sont de taille 4.

Cette précédente définition de PLSC à deux ordres à *buckets* permet d'ordonner des éléments alors que ceux-ci sont dans le même *bucket* dans chacun des ordres de départ, c'est-à-dire incomparables dans chacun de ces ordres, *e.g.* les éléments d, e et f dans les PLSC l'exemple précédent. Dans le contexte de la comparaison de cartes génomiques, cela a pour conséquence qu'une PLSC à deux cartes ordonne deux éléments alors qu'aucune des deux cartes en entrée ne les ordonnait. C'est pour cette raison que nous proposons une seconde définition d'une sous-séquence commune à deux ordres à *buckets*, plus stricte mais plus réaliste (puisque n'inférant pas de relation qui n'est présente dans aucun des ordres de départ), que nous nommons *sous-séquence commune induite*.

Définition 13 (Sous-séquence commune induite et PLSCI à deux ordres à *buckets*)**.** *Soient π_1 et π_2 des ordres à buckets de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$. Une plus longue sous-séquence commune induite (PLSCI) à π_1 et π_2 est une sous-séquence $s = (e_1, \ldots, e_l)$ de π_1 et de π_2 qui est telle que $\forall 1 \leq i < j \leq l$, soit $e_i \prec_{\pi_1} e_j$ et $e_j \not\prec_{\pi_2} e_i$, soit $e_i \prec_{\pi_2} e_j$ et $e_j \not\prec_{\pi_1} e_i$.*

Une PLSCI à deux ordres à *buckets* π_1 et π_2 ne peut donc pas contenir plusieurs éléments situés dans le même *bucket* de π_1 et dans le même *bucket* de π_2.

Exemple 8. *Soient π_1 et π_2 de l'exemple précédent.*

Les suites (a, b), (g, h), (h, g), (c, e), (c, d, e, f), (c, e, f, d) sont des sous-séquences communes induites à π_1 et π_2 (il en existe d'autres). Il existe quatre PLSCI à π_1 et π_2 : (a, b), (c, d), (c, e) et (c, f), qui sont de taille 2.

Remarque 5. *Les ensembles des PLSC et des PLSCI à deux ordres à buckets peuvent être disjoints, chevauchants, ou égaux.*

Afin de montrer le corollaire 1, qui garantit que les PLSC à deux ordres à *buckets* sont les mêmes que celles à leurs ordres homogénéisés, nous donnons d'abord un ensemble de propriétés sur ces PLSC.

Propriété 4. *Soit s une PLSC à deux ordres à buckets π_1 et π_2. Si s contient un élément e, alors elle contient tous les éléments du bucket B de π_1^h et π_2^h (les ordres issus de l'homogénéisation de π_1 et π_2) qui contient e.*

Démonstration. Évident (sinon on pourrait agrandir s). □

Lemme 2. *Soient π_1 et π_2 deux ordres à buckets de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$, et π_1^h et π_2^h les ordres à buckets respectifs issus de leurs homogénéisations. Une sous-séquence commune à π_1^h et π_2^h est aussi une sous-séquence commune à π_1 et π_2.*

Démonstration. Par l'absurde. Soit s une sous-séquence commune à π_1^h et π_2^h. Supposons que s n'est pas une sous-séquence commune à π_1 et π_2. Sans perte de généralité, supposons que s n'est pas une sous-séquence de π_1. Par conséquent, il existe deux éléments successifs $s[i]$ et $s[i+1]$ de s tels que $s[i+1] \prec_{\pi_1} s[i]$. Du coup, il existe B_k et B_l des *buckets* de π_1 tels que B_k est placé avant B_l, et $s[i+1] \in B_k$ et $s[i] \in B_l$. Dès lors, $pos_1(s[i+1]) < pos_1(s[i])$, et par construction, $s[i+1] \prec_{\pi_1^h} s[i]$: c'est une contradiction. s est donc bien une sous-séquence de π_1. On montrerait de même que s est bien une sous-séquence de π_2. Du coup, s est une sous-séquence commune à π_1 et π_2. □

Théorème 1. *L'ensemble des sous-séquences communes à deux ordres à buckets π_1 et π_2 est égal à l'ensemble des sous-séquences communes aux deux ordres à buckets π_1^h et π_2^h issus respectivement de l'homogénéisation de π_1 et π_2.*

Démonstration. $\subseteq$ Par l'absurde. Sans perte de généralité, supposons que s n'est pas une sous-séquence de π_1^h. Comme tous les éléments de s sont présents dans π_1 et dans π_2, ils appartiennent à $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$ et donc par définition à π_1^h. Par conséquent, il existe deux éléments successifs $s[i]$ et $s[i+1]$ de s tels que $s[i+1] \prec_{\pi_1^h} s[i]$. Du coup, il existe B_k et B_l des *buckets* de π_1^h tels que B_k est placé avant B_l, et $s[i+1] \in B_k$ et $s[i] \in B_l$. Trois cas sont possibles :

- $s[i]$ et $s[i+1]$ appartiennent au même *bucket* dans π_1 et dans π_2. Du coup, $pos_1(s[i]) = pos_1(s[i+1])$ et $pos_2(s[i]) = pos_2(s[i+1])$. Par construction, $s[i]$ et $s[i+1]$ appartiennent aussi au même *bucket* dans π_1^h : c'est une contradiction.

- $s[i]$ et $s[i+1]$ appartiennent à des *buckets* différents dans π_1. Du coup, $pos_1(s[i]) < pos_1(s[i+1])$ car s est une sous-séquence de π_1. Par construction, dans π_1^h, $s[i]$ appartient à un *bucket* situé avant le *bucket* contenant $s[i+1]$: c'est une contradiction.

- $s[i]$ et $s[i+1]$ appartiennent au même *bucket* dans π_1 et à des *buckets* différents dans π_2. Du coup, $pos_1(s[i]) = pos_1(s[i+1])$ et $pos_2(s[i]) < pos_2(s[i+1])$ car s est une sous-séquence de π_2. Par construction, dans π_1^h, $s[i]$ appartient à un *bucket* situé avant le *bucket* contenant $s[i+1]$: c'est une contradiction.

Comme tous les cas possibles mènent à une contradiction, l'hypothèse de départ est fausse, et donc s est bien une sous-séquence de π_1^h. On montrerait de même que s est bien une sous-séquence de π_2^h.

$\supseteq$ Lemme 2. □

Le corollaire suivant est obtenu par le théorème 1.

Corollaire 1. *Soient deux ordres à buckets π_1 et π_2, ainsi que π_1^h et π_2^h leurs homogénéisations respectives. L'ensemble des PLSC à π_1 et π_2 est égal à l'ensemble des PLSC à π_1^h et π_2^h.*

Montrons maintenant le théorème suivant, qui permet par la suite au corollaire 2 de garantir que les PLSCI à deux ordres à *buckets* sont les mêmes que celles à leurs ordres homogénéisés.

Théorème 2. *L'ensemble des sous-séquences communes induites à deux ordres à buckets π_1 et π_2 est égal à l'ensemble des sous-séquences communes induites aux deux ordres à buckets π_1^h et π_2^h issus respectivement de l'homogénéisation de π_1 et π_2.*

Démonstration. $\subseteq$ Soit s une sous-séquence commune induite à π_1 et π_2. Tout d'abord, s est une sous-séquence commune à π_1^h et π_2^h (théorème 1). De plus, par définition d'une sous-séquence commune induite : $\forall 1 \leq i < j \leq l$: soit i) $s[i] \prec_{\pi_1} s[j]$ soit ii) $s[i] \prec_{\pi_2} s[j]$. Dans le cas i) $s[i]$ est dans un *bucket* précédant $s[j]$ dans π_1^h. Dans le cas ii) $s[i]$ est dans un *bucket* précédant $s[j]$ dans π_2^h. Dans les deux cas, $s[i]$ est dans un *bucket* situé le *bucket* contenant $s[j]$ dans au moins un des deux ordres π_1^h et π_2^h. Par conséquent, s est aussi une sous-séquence commune induite à π_1^h et π_2^h.

$\supseteq$ Soit s une sous-séquence commune induite à π_1^h et π_2^h. D'abord, s est une sous-séquence commune à π_1 et π_2 (théorème 1). De plus, par définition d'une sous-séquence commune induite : $\forall 1 \leq i < j \leq l$: soit i) $s[i] \prec_{\pi_1^h} s[j]$ soit ii) $s[i] \prec_{\pi_2^h} s[j]$. Dans le cas i) $s[i]$ est dans un *bucket* précédant $s[j]$ dans π_1^h implique que soit a) $s[i] \prec_{\pi_1} s[j]$, soit b) ($s[i] \not\prec_{\pi_1} s[j]$ et $s[i] \prec_{\pi_2} s[j]$). Dans les deux cas a) et b), $s[i]$ est dans un *bucket* situé le *bucket* contenant $s[j]$ dans au moins un des deux ordres π_1 et π_2. De manière similaire, dans le cas ii) $s[i]$ est dans un *bucket* précédant $s[j]$ dans π_2^h implique soit a) $s[i] \prec_{\pi_2} s[j]$, soit b) ($s[i] \not\prec_{\pi_2} s[j]$ et $s[i] \prec_{\pi_1} s[j]$). Dans les deux cas a) et b), $s[i]$ est dans un *bucket* situé le *bucket* contenant $s[j]$ dans au moins un des deux ordres π_1 et π_2. Par conséquent, s est aussi une sous-séquence commune induite à π_1 et π_2.

□

Le corollaire suivant est obtenu par le théorème 2.

Corollaire 2. *Soient deux ordres à buckets π_1 et π_2, ainsi que π_1^h et π_2^h leurs homogénéisations respectives. L'ensemble des PLSCI à π_1 et π_2 est égal à l'ensemble des PLSCI à π_1^h et π_2^h.*

Maintenant que nous avons défini le problème de PLSC(I) à deux ordres à *buckets*, nous détaillons, dans les deux sous-sections suivantes, deux manières d'obtenir une PLSC à deux ordres à *buckets*. La première (section 4.2) est basée sur un algorithme de programmation dynamique, alors que la deuxième (section 4.3) sur le calcul d'une plus longue sous-séquence augmentante.

4.2 PLSC à deux ordres à *buckets* par un algorithme de programmation dynamique

Étant donné deux ordres totaux τ_1 et τ_2 de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$, il existe un algorithme de programmation dynamique $\mathcal{A}$ [12] permettant de retourner une PLSC(I) à τ_1 et τ_2 en temps $\mathcal{O}(|\mathcal{D}|^2)$, avec $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$. Dans cette section, nous adaptons cet algorithme $\mathcal{A}$ au calcul d'une PLSC(I) à deux ordres à *buckets*. Pour cela, nous proposons un premier algorithme, l'algorithme 3 page 21, qui étant donnés deux ordres à *buckets* π_1 et π_2, rempli, selon la formule 4.1, la matrice L des longueurs des PLSC(I) (définition 14). Ensuite, nous donnons un deuxième algorithme, l'algorithme 4 page 24, qui étant donné π_1, π_2 et L, retourne une PLSC(I) à π_1 et π_2, en réalisant un *backtrack* dans la matrice L.

Définition 14 (Matrice des longueurs des PLSC (resp. PLSCI))**.** *Soit π_1 et π_2 deux ordres à buckets, et π_1^h et π_2^h les ordres à buckets respectifs issus de leurs homogénéisations. La matrice L des longueurs des PLSC (resp. PLSCI) à deux ordres à buckets π_1 et π_2 est la matrice d'entiers de taille $(k+1) \times (k+1)$, avec k le nombre de buckets de π_1^h et de π_2^h, telle que $L[i,j]$ est la taille d'une PLSC (resp. PLSCI) à l'ordre à buckets induit par les i premiers buckets de π_1^h, et à l'ordre à buckets induit par les j premiers buckets de π_2^h.*

La formule suivante [12] permet de remplir la matrice L des longueurs des PLSC(I) à deux ordres à *buckets* π_1^h et π_2^h issus d'une homogénéisation :

$$L[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ L[i-1, j-1] + x & \text{si le } i^{\text{e}} \textit{ bucket} \text{ B de } \pi_1^h \text{ est égal au } j^{\text{e}} \textit{ bucket} \text{ de } \pi_2^h \\ max\{L[i-1,j], L[i,j-1]\} & \text{sinon,} \end{cases} \tag{4.1}$$

avec $x = |B|$ pour la matrice des longueurs des PLSC, et $x = 1$ pour la matrice des longueurs des PLSCI.

Exemple 9. *Soient*
$\pi_1^h = (\{g,h\},\{c\},\{d,e,f\},\{a\},\{b\})$ *et*
$\pi_2^h = (\{a\},\{b\},\{c\},\{d,e,f\},\{g,h\})$ *deux ordres à buckets issus d'une homogénéisation. La matrice des longueurs des PLSC (resp. PLSCI) à* π_1^h *et* π_2^h *est la suivante, dont les valeurs des cases sont en bleu (resp. en violet) :*

L		$\{a\}$	$\{b\}$	$\{c\}$	$\{d,e,f\}$	$\{g,h\}$
	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
$\{g,h\}$	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	2 / 1
$\{c\}$	0 / 0	0 / 0	0 / 0	1 / 1	1 / 1	2 / 1
$\{d,e,f\}$	0 / 0	0 / 0	0 / 0	1 / 1	4 / 2	4 / 2
$\{a\}$	0 / 0	1 / 1	1 / 1	1 / 1	4 / 2	4 / 2
$\{b\}$	0 / 0	1 / 1	2 / 2	2 / 2	4 / 2	4 / 2

Dans cette matrice L*, la case* $L[k+1,k+1]$*, avec* k *le nombre de buckets de* π_1^h *et* π_2^h*, indique la longueur d'une PLSC(I) à* π_1^h *et* π_2^h*. Dans cet exemple, la taille d'une PLSC à* π_1^h *et* π_2^h *est* $L[6,6] = 4$*, et la taille d'une PLSCI à* π_1^h *et* π_2^h *est* $L[6,6] = 2$*.*

L'algorithme 3 page 21 permet de remplir la matrice des longueurs des PLSC(I) à deux ordres à *buckets*, selon la formule 4.1. Dans cet algorithme, nous supposons que les éléments d'un *bucket* sont triés selon un ordre total sur $\mathcal{D}$, ce qui permet de savoir si deux *buckets* sont identiques en regardant seulement leurs premiers éléments (ligne 7 de l'algorithme 3 page 21).

Proposition 7 (Terminaison de l'algorithme 3)**.** *L'algorithme 3 termine.*

Démonstration. Toutes les boucles **"pour"** itèrent sur des éléments finis. □

Proposition 8 (Correction de l'algorithme 3)**.** *L'algorithme 3 rempli la matrice des longueurs des PLSC(I).*

Démonstration. Il s'agit d'un algorithme classique proposé dans [12]. □

Proposition 9 (Complexité en temps de l'algorithme 3)**.** *L'algorithme 3 nécessite* $\mathcal{O}(|\mathcal{D}|^2)$ *en temps.*

Démonstration. La boucle ligne 2 est en $\mathcal{O}(k)$, avec $k \leq |\mathcal{D}|$ et $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$. Les boucles lignes 5 et 6 sont en $\mathcal{O}(k^2)$. □

Afin d'obtenir une PLSC(I) s, il est possible de réaliser un *backtrack* dans la matrice des longueurs des PLSC(I). Pour cela, il suffit de suivre un "bon" chemin dans la matrice en partant de la case $L[k,k]$, avec k le nombre de *buckets* des deux ordres.

Algorithme 3 : MATRICE DES LONGUEURS DES PLSC(I)

Données : Deux ordres à *buckets* π_1^h et π_2^h non vides d'un même domaine $\mathcal{D}$ issus d'une homogénéisation.
Un booléen *induite*, valant vrai si on calcule la matrice des longueurs des PLSC(I), et faux si on calcule la matrice des longueurs des PLSC.

Résultat : L, la matrice des longueurs des PLSC(I) à π_1^h et π_2^h.

1 Soient $\mathcal{B}^{h_1} = (B_1^{h_1}, \ldots, B_k^{h_1})$ et $\mathcal{B}^{h_2} = (B_1^{h_2}, \ldots, B_k^{h_2})$ les suites respectives des *buckets* de π_1^h et π_2^h, ainsi que L une matrice d'entiers de taille $(k+1) \times (k+1)$.

2 **pour** i *de* 0 *à* k **faire**
3 $\quad L[i, 0] \leftarrow 0$;
4 $\quad L[0, i] \leftarrow 0$;

5 **pour** i *de* 1 *à* k **faire** // i parcourt les *buckets* de π_1^h
6 $\quad$ **pour** j *de* 1 *à* k **faire** // j parcourt les *buckets* de π_2^h
$\quad\quad$ // Éléments d'un *bucket* ordonnés selon un ordre total sur $\mathcal{D}$
7 $\quad\quad$ **si** $B_i^{h_1}[1] = B_j^{h_2}[1]$ **alors**
8 $\quad\quad\quad$ **si** $induite = vrai$ **alors**
9 $\quad\quad\quad\quad L[i, j] \leftarrow L[i-1, j-1] + 1$;
10 $\quad\quad\quad$ **sinon**
11 $\quad\quad\quad\quad L[i, j] \leftarrow L[i-1, j-1] + |B_i|$;
12 $\quad\quad$ **sinon**
13 $\quad\quad\quad L[i, j] \leftarrow max\{L[i, j-1], L[i-1, j]\}$;

14 **retourner** L ;

Si la PLSC(I) s contient un élément d'un *bucket* B, qui est le i^{e} *bucket* de π_1^h et le j^{e} *bucket* de π_2^h, alors le chemin suivi dans la matrice pour obtenir s passe par la case $L[i, j]$.

De plus, la construction d'un tel chemin est la suivante : lorsque le chemin est construit (depuis $L[k, k]$) jusqu'à la case $L[i, j]$, pour connaître la (les) case(s) suivante(s) du chemin, il faut savoir comment a été calculé $L[i, j]$:

- si $L[i, j]$ a été obtenu depuis $L[i-1, j-1]$, alors la case suivante dans le chemin est $L[i-1, j-1]$ (et les éléments (resp. exactement un élément) du i^{e} *bucket* de π_1^h sont contenus (resp. est contenu) dans la PLSC (resp. PLSCI) s),
- sinon, la case suivante est la case qui contient le nombre le plus grand parmi $L[i, j-1]$ et $L[i-1, j]$ (si $L[i, j-1] = L[i-1, j]$, alors le chemin a le choix entre les deux cases).

Exemple 10. *Soient π_1^h, π_2^h et L (la matrice des longueurs des PLSC à π_1^h et π_2^h) de l'exemple précédent. Ci-dessous est illustré L, dans laquelle les cases $L[i, j]$ ont été enrichies d'une ou de plusieurs flèches (sauf pour la première ligne et la première colonne), indiquant depuis quelle case a été calculé $L[i, j]$ dans la matrice des PLSC. De plus, les flèches rouge indiquent un backtrack réalisé dans la matrice : elles induisent un "bon" chemin dans la matrice, qui permet de construire l'ensemble de PLSC $L(\{\{c\}, \{d, e, f\}\})$:* (c, d, e, f), (c, d, f, e), (c, e, d, f), (c, e, f, d), (c, f, d, e), *et* (c, f, e, d).

L		$\{a\}$	$\{b\}$	$\{c\}$	$\{d, e, f\}$	$\{g, h\}$
	0	0	0	0	0	0
$\{g, h\}$	0	$\leftarrow\uparrow 0$	$\leftarrow \uparrow 0$	$\leftarrow\uparrow 0$	$\leftarrow\uparrow 0$	$\nwarrow 2$
$\{c\}$	0	$\leftarrow\uparrow 0$	$\leftarrow\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\uparrow 2$
$\{d, e, f\}$	0	$\leftarrow\uparrow 0$	$\leftarrow\uparrow 0$	$\uparrow 1$	$\nwarrow 4$	$\leftarrow 4$
$\{a\}$	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow\uparrow 1$	$\uparrow 4$	$\leftarrow \uparrow 4$
$\{b\}$	0	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 4$	$\leftarrow \uparrow 4$

Un même backtrack dans la matrice des longueurs des PLSCI (non montré ici) permettrait de construire une PLSCI, en récupérant exactement un élément par bucket B, égal au i^{e} bucket de π_1^h et au j^{e} bucket de π_2^h, lorsque le chemin passe par la case $L[i, j]$. Cependant, les flèches ne sont pas les mêmes dans la matrice des longueurs des PLSC et dans celle des PLSCI (puisque les valeurs dans la matrice des longueurs des PLSC sont indépendantes de celles dans la matrice des longueurs des PLSCI).

L'algorithme 4 page 24, toujours adapté de [12], utilise la matrice des longueurs des PLSC(I) pour retourner, grâce à un *backtrack* dans cette dernière, une PLSC(I) à deux ordres à *buckets* donnés.

Proposition 10 (Terminaison de l'algorithme 4)**.** *L'algorithme 4 termine.*

Démonstration. La boucle **"tant que"** est finie : à chaque itération, au moins une des variables i ou j est décrémentée de 1. La boucle **"pour"** itère sur des éléments finis. □

Proposition 11 (Correction de l'algorithme 4)**.** *L'algorithme 4 rempli la matrice des longueurs des PLS(B)C.*

Démonstration. Il s'agit d'un algorithme classique proposé dans [12]. □

Proposition 12 (Complexité en temps de l'algorithme 4)**.** *L'algorithme 4 est en temps* $\mathcal{O}(|\mathcal{D}|)$.

Démonstration. La boucle ligne 5 s'exécute au plus $2k+1$ fois (si les variables j et i sont toujours décrémentées par les lignes 14 et 15 respectivement), avec $k \leq |\mathcal{D}|$ le nombre de *buckets* des ordres issus de l'homogénéisation des ordres donnés. L'instruction de la ligne 10 peut s'implémenter en temps constant si l'ordre π en construction est géré par une liste de *buckets* ; cette ligne s'exécute au plus $|\mathcal{D}|$ fois (complexité amortie). □

Une PLSC(I) à deux ordres à *buckets* π_1 et π_2 de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$ peut donc être obtenue par la procédure suivante :

1. Appels de l'algorithme 1 pour obtenir les ordres homogénéisés π_1^h et π_2^h :

$$\pi_2^h \leftarrow \textsc{Homogénéisation}(\pi_1, \pi_2)$$

$$\pi_1^h \leftarrow \textsc{Homogénéisation}(\pi_2, \pi_1)$$

2. Appel de l'algorithme 3 pour obtenir la matrice L des longueurs des PLSC(I) à π_1 et π_2 :

$$L \leftarrow \textsc{Matrice des longueurs des PLSC(I)}(\pi_1^h, \pi_2^h)$$

3. Appel de l'algorithme 4 pour récupérer une PLSC(I) τ à π_1 et π_2 :

$$\tau \leftarrow \textsc{Backtrack matrice des longueurs des PLSC(I)}(\pi_1^h, \pi_2^h, L)$$

Algorithme 4 : BACKTRACK MATRICE DES LONGUEURS DES PLSC(I)

Données : Deux ordres à *buckets* π_1^h et π_2^h non vides d'un même domaine $\mathcal{D}$ issus d'une homogénéisation.
L, la matrice des longueurs des PLSC(I) à π_1^h et π_2^h.
Un booléen *induite*, valant vrai si on cherche une PLSCI, et faux si on cherche une PLSC.

Résultat : Une PLSC(I) à π_1^h et π_2^h.

Soient $\mathcal{B}^{h_1} = (B_1^{h_1}, \ldots, B_k^{h_1})$ et $\mathcal{B}^{h_2} = (B_1^{h_2}, \ldots, B_k^{h_2})$ les suites respectives des *buckets* de π_1^h et π_2^h.

Soit τ un ordre vide.

$i \leftarrow k$;
$j \leftarrow k$;

tant que $i > 0$ *et* $j > 0$ **faire**

  si $B_i^{h_1}[1] = B_j^{h_2}[1]$ **alors**
    `// Ajout des éléments de` $B_i^{h_1}$ `à la suite de` τ.
    si *induite* $= vrai$ **alors**
      $\tau.push\_front(B_i^{h_1}[1])$;
    sinon
      pour e *de* $B_i^{h_1}[1]$ *à* $B_i^{h_1}[|B_i^{h_1}|]$ **faire** $\tau.push\_front(e)$;
    $i--$;
    $j--$;
  sinon
    si $L[i, j-1] > L[i-1, j]$ **alors** $j--$;
    sinon $i--$;

retourner τ ;

Sans perte de généralité, disons que $|\mathcal{D}_1| \leq |\mathcal{D}_2|$. La complexité en temps de cette procédure est $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|) + |\mathcal{D}_1|log(|\mathcal{D}_2|) + |\mathcal{D}|^2)$ (voir propositions 3 page 11, 9 page 20 et 12 page 23), avec $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$.

Dans la section suivante, nous détaillons un nouvel algorithme, basé sur le calcul d'une plus longue sous-séquence augmentante, permettant d'obtenir une PLSC à π_1 et π_2 et ayant une complexité plus faible que la procédure que nous venons de voir.

4.3 PLSC à deux ordres à *buckets* par une plus longue sous-séquence augmentante (PLSA)

Dans un premier temps, nous montrons dans l'exemple 11 comment passer par le calcul d'une PLSA pour l'obtention d'une PLSCI à deux ordres à *buckets*. Les algorithmes donnés dans un second temps permettent de retourner une PLSC à deux ordres à *buckets*, selon le même principe.

Exemple 11. *Soient*
$\pi_1^h = (\{g,h\},\{c\},\{d,e,f\},\{a\},\{b\})$ *et*
$\pi_2^h = (\{a\},\{b\},\{c\},\{d,e,f\},\{g,h\})$ *deux ordres à buckets issus d'une homogénéisation.*

*Soit buc*1*posBuc*2 *un tableau d'entiers de taille k tel que la* i^e *case contient l'entier j si, et seulement si, le* i^e *bucket de* π_1^h *est égal au* j^e *bucket de* π_2^h*. Dans cet exemple,*

*buc*1*posBuc*2 $=$ | 5 | 3 | 4 | 1 | 2 |. *Remarquons qu'à une PLSA* $(s_1,\ldots,s_l)$ *de ce tableau, e.g.* $s = (3,4)$*, est associé un ensemble de PLSCI à* π_1^h *et* π_2^h*, en récupérant exactement un élément dans chaque* $s_i{}^e$ *bucket de* π_2^h*,* $1 \leq i \leq l$*. Les PLSCI associées à s sont donc* (c,d)*,* (c,e) *et* (c,f)*. L'autre plus longue sous-séquence augmentante de cet exemple est* $(1,2)$*, à laquelle est associée une seule PLSCI :* (a,b)*.*

Pour adapter cette idée au calcul d'une PLSC à deux ordres à *buckets* π_1 et π_2 (d'homogénéisation π_1^h et π_2^h respectivement), il suffit de passer d'abord par une des extensions linéaires de π_1^h et une des extensions linéaires de π_2^h, telles que deux mêmes *buckets* soient étendus de la même manière dans ces deux extensions (*e.g.* sur la base d'un ordre total sur $\mathcal{D}$). Nous illustrons un tel calcul dans l'exemple suivant.

Exemple 12. *Soient*
$\pi_1^h = (\{g,h\},\{c\},\{d,e,f\},\{a\},\{b\})$ *et*
$\pi_2^h = (\{a\},\{b\},\{c\},\{d,e,f\},\{g,h\})$ *deux ordres à buckets sur* $\mathcal{D}$ *issus d'une homogénéisation.*

*Soient ext*1 *et ext*2 *des extensions linéaires de* π_1^h *et* π_2^h *respectivement, qui linéarisent les buckets selon un ordre total sur* $\mathcal{D}$*.*

*Soit ext*1*posExt*2 *un tableau d'entiers de couples* (e,pos)*,* $e \in \mathcal{D}$*,* $pos \in \mathbb{N}$*, qui est tel que la case i contient le couple* (e,pos) *si, et seulement si, e est le* i^e *élément de ext*1 *et e est le* pos^e *élément de ext*2*. Dans cet exemple, si l'ordre total sur* $\mathcal{D}$ *est l'ordre lexicographique,*

$ext1posExt2 =$ | $(g,7)$ | $(h,8)$ | $(c,3)$ | $(d,4)$ | $(e,5)$ | $(f,6)$ | $(a,1)$ | $(b,2)$ |.

Remarquons qu'à une PLSA $s = (s_1, \ldots, s_l) = (3,4,5,6)$ *des secondes positions ce tableau est associé un ensemble de PLSC à* π_1^h *et* π_2^h*, en linéarisant chaque* ${s_i}^e$ *bucket de* π_2^h*,* $1 \leq i \leq l$*, une seule fois (dans cet exemple les* ${s_2}^e$*,* ${s_3}^e$ *et* ${s_4}^e$ *buckets sont les mêmes). Les PLSC associées à* s *sont donc* (c,d,e,f)*,* (c,d,f,e)*,* (c,e,f,d)*,* (c,e,d,f)*,* (c,f,e,d) *et* (c,f,d,e)*.*

L'algorithme 5 page 27 nous permet d'obtenir une structure de données sur laquelle le calcul d'une PLSA produira une PLSC. Plus précisément, cet algorithme retourne, étant donnés deux ordres à *buckets* π_1^h et π_2^h issus d'une homogénéisation, un tableau $ext1posExt2$ de taille $|\mathcal{D}|$ de couples d'éléments (e, pos), $e \in \mathcal{D}$, $pos \in \mathbb{N}$: la suite des $e \in (e, pos)$ de $ext1posExt2[1]$ à $ext1posExt2[|\mathcal{D}|]$ représente l'ordre des éléments de $\mathcal{D}$ dans une extension linéaire $ext1$ de π_1^h, et pos représente le numéro de l'élément e dans l'ordre d'une extension linéaire $ext2$ de π_2^h. De plus, $ext1$ et $ext2$ linéarisent de la même manière deux *buckets* identiques.

Proposition 13 (Terminaison de l'algorithme 5)**.** *L'algorithme 5 termine.*

Démonstration. Toutes les boucles *"pour"* itèrent sur des éléments finis. □

Proposition 14 (Correction de l'algorithme 5)**.** *L'algorithme 5 retourne, pour les deux ordres à buckets* π_1^h *et* π_2^h *sur* $\mathcal{D}$ *issus d'une homogénisation qui lui sont donnés en entrée, un tableau* $ext1posExt2$ *de taille* $|\mathcal{D}|$ *de couples d'éléments* (e, pos)*,* $e \in \mathcal{D}$*,* $pos \in \mathbb{N}$*, tel que :*

- *la suite des* e *de* $ext1posExt2[1].first$ *à* $ext1posExt2[|\mathcal{D}|].first$ *représente l'ordre des éléments de* $\mathcal{D}$ *dans une extension linéaire* $ext1$ *de* π_1^h*,*
- *Les* pos *de chaque* $(e, pos) \in ext1posExt2$ *représentent la position de l'élément* e *dans une extension linéaire* $ext2$ *de* π_2^h*.*

Démonstration. La ligne 3 parcourt les *buckets* de π_2^h, de $B_1^{h_2}$ à $B_k^{h_2}$. Pour chacun de ces *buckets* B_i, la ligne 4 itère sur ses éléments e, de $B_i[1]$ à $B_i[|B_i|]$. Pour chacun de ces éléments e, la ligne 5 ajoute, dans le dictionnaire de hashage $dicPosExt2$, la clé e correspondant à la valeur $posExt2$ si, et seulement si, e est le ${posExt2}^{\text{e}}$ sommet à être traité.

Les boucles lignes 9 et 10 parcourent de la même façon les éléments de π_1^h. Pour chaque élément e de π_1^h visité à l'itération $posExt1$ de ce parcours, la ligne 12 ajoute $(e, dicPosExt2.getValue(e))$ à la ${posExt1}^{\text{e}}$ case du tableau $ext1posExt2$. □

Proposition 15. *L'algorithme 5 nécessite* $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$ *en temps.*

Démonstration. Les boucles imbriquées lignes 3 et 4 parcourent les éléments de π_2, qui sont du nombre de $|\mathcal{D}|$. Aussi, une insertion dans le dictionnaire de hashage se réalise en un temps logarithmique en son nombre d'éléments, c'est-à-dire $\mathcal{O}(log(|\mathcal{D}|))$. Ces boucles s'exécutent donc en un temps total de $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$.

Algorithme 5 : CODAGE EXTENSIONS LINÉAIRES

Données : Deux ordres à *buckets* π_1^h et π_2^h non vides d'un même domaine $\mathcal{D}$ issus d'une homogénéisation, qui sont tels que deux *buckets* identiques soient ordonnés de la même manière.

Résultat : $ext1posExt2$, un tableau de taille $|\mathcal{D}|$ de couples d'éléments (e, pos), $e \in \mathcal{D}$, $pos \in \mathbb{N}$, tel que :

- la suite des e de $ext1posExt2[1].first$ à $ext1posExt2[|\mathcal{D}|].first$ représente l'ordre des éléments de $\mathcal{D}$ dans une extension linéaire $ext1$ de π_1^h,
- Les pos de chaque $(e, pos) \in ext1posExt2$ représentent la position de l'élément e dans une extension linéaire $ext2$ de π_2^h.

Soients $\mathcal{B}^{h_1} = (B_1^{h_1}, \ldots, B_k^{h_1})$ et $\mathcal{B}^{h_2} = (B_1^{h_2}, \ldots, B_k^{h_2})$ les suites respectives des *buckets* de π_1^h et π_2^h, ainsi que $dicPosExt2$ un dictionnaire de hashage vide.

$posExt2 \leftarrow 1$;
pour B_i *de* $B_1^{h_2}$ *à* $B_k^{h_2}$ **faire**
    pour e *de* $B_i[1]$ *à* $B_i[|B_i|]$ **faire**
        $dicPosExt2.insert(key = e, value = posExt2)$;
        $posExt2++$;

Soit $ext1posExt2$ un tableau vide de taille $|\mathcal{D}|$ de couples d'éléments (e, pos), $e \in \mathcal{D}$, $pos \in \mathbb{N}$.

$posExt1 \leftarrow 1$;
pour B_i *de* $B_1^{h_1}$ *à* $B_k^{h_1}$ **faire**
    pour e *de* $B_i[1]$ *à* $B_i[|B_i|]$ **faire**
        $posExt2 \leftarrow dicPosExt2.getValue(e)$;
        $ext1posExt2[posExt1] \leftarrow (e, posExt2)$;
        $posExt1++$;

retourner $ext1posExt2$;

Les boucles lignes 9 et 10 sont en $\mathcal{O}(|\mathcal{D}|)$. L'ensemble des récupérations de valeurs dans le dictionnaire de hashage réalisées à la ligne 11 est en $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$. Finalement, comme les autres lignes s'exécutent en temps constant, l'algorithme 5 est en $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$ dans le pire des cas. □

Nous présentons maintenant l'algorithme 6 page 31 qui, en passant par le calcul d'une PLSA, retourne une PLSC à deux ordres à *buckets* issus d'une homogénéisation. L'algorithme que nous utilisons pour le calcul d'une PLSA d'une suite d'entiers de taille n est connu de la littérature : il est donné en annexe A page 37. Sa complexité en temps est $\mathcal{O}(nlog(n))$ [9, 8], où n est le nombre d'éléments de la séquence passée en entrée.

Proposition 16 (Terminaison de l'algorithme 6)**.** *L'algorithme 6 termine.*

Démonstration. Toutes les boucles **"pour"** itèrent sur des éléments finis. La boucle **"tant que"** ligne 11 cherche, pour chaque élément $L[i]$ de la liste L, la case j du tableau $ext1posExt2$ qui contient $L[i]$ en deuxième position. Comme la liste L ne contient que des éléments présents dans tab, qui est une copie des secondes positions de $ext1posExt2$, la case j cherchée existe toujours. De plus, il n'est pas nécessaire de commencer à chercher cette case depuis le début de la liste, mais seulement depuis l'indice suivant celui qui a rendu la précédente recherche fructueuse. En effet, L correspond à une sous-séquence des secondes positions de $ext1posExt2$: une fois un élément $L[i]$ trouvé, le prochain qui sera cherché sera forcément situé après $L[i]$ dans L. □

Proposition 17 (Correction de l'algorithme 6)**.** *L'algorithme 6 retourne une liste représentant une PLSC à π_1^h et π_2^h, les deux ordres à buckets issus d'une homogénéisation qui lui sont donnés en entrée.*

Démonstration. La ligne 4 fait une copie des secondes positions de $ext1posExt2$ dans le tableau tab. Du coup, L contient une PLSA $(n_1, \ldots, n_l)$ de la séquence $ext1posExt2[1][2] \ldots ext1posExt2[|\mathcal{D}|][2]$ formée par les secondes positions du tableau $ext1posExt2$ (correction de l'algorithme 7 en annexe page 37). Soit $I(n_i)$ l'indice de $ext1posExt2$ qui contient n_i en seconde position.

Montrons que la suite $s = (ext1posExt2[I(n_1)][1], \ldots, ext1posExt2[I(n_l)][1])$ est $(\star)$ une sous-séquence de π_2^h et de π_1^h qui est $(\star\star)$ de taille maximale, *i.e.*, une PLSC à π_2^h et π_1^h.

$(\star)$ Comme les secondes positions pos de chaque $(e, pos) \in ext1posExt2$ représentent la position de l'élément e dans une extension linéaire $ext2$ de π_2^h (proposition 14) et que $(ext1posExt2[I(n_1)][2], \ldots, ext1posExt2[I(n_l)][2])$ est une PLSA de $ext1posExt2[1][2] \ldots ext1posExt2[|\mathcal{D}|][2]$, la suite s est bien une sous-séquence de π_2^h. Par ailleurs, puisque la suite des e de $ext1posExt2[1][1]$ à $ext1posExt2[|\mathcal{D}|][1]$ représente l'ordre des éléments de $\mathcal{D}$ dans une extension linéaire $ext1$ de π_1^h, la suite s est bien une sous-séquence de π_1^h.

$(\star\star)$ Par l'absurde. Supposons qu'il existe une sous-séquence s' commune à π_1^h et π_2^h, avec $|s'| > |s|$. Par la propriété 4, pour tout élément s'_i de s' appartenant au *bucket* B_α de π_1^h et π_2^h, tous les éléments de B_α appartiennent à s'. Sans perte de généralité, supposons que pour chaque élément s'_i de s' appartenant au *bucket* B_α, l'ensemble des éléments de

B_α apparaissent dans s' dans le même ordre que celui défini par les extensions linéaires de π_1^h et π_2^h dans l'algorithme 5 (pour rappel, ces extensions linéarisent de la même façon des *buckets* identiques). Soit $(B_1, \ldots, B_c)$ la suite maximale des *buckets* de π_1^h et π_2^h qui ont leurs éléments dans s', et telle que $\forall 1 \leq i < j \leq c$, $\forall e \in B_i$, $\forall e' \in B_j$, $e \prec_{\pi_1^h} e'$ et $e \prec_{\pi_2^h} e'$. Puisque chacun de ces *buckets* a été linéarisé par l'algorithme 5 dans le même ordre que l'ordre d'apparition de leurs éléments dans s', l'appel de l'algorithme 7 réalisé par la ligne 6 de l'algorithme 6 aurait retourné une PLSA associée à s' (et donc à la suite $(B_1, \ldots, B_c)$) de taille supérieure à celle associée à s. Il s'agit d'une contradiction. □

Proposition 18. *L'algorithme 6 nécessite $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$ en temps.*

Démonstration. L'appel (ligne 2) de l'algorithme 5 (CODAGE EXTENSIONS LINÉAIRES) est en $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$ (voir proposition 15 page 26). La boucle de la ligne 4 s'exécute au plus $|\mathcal{D}|$ fois. De plus, l'appel (ligne 6) de l'algorithme 7 (PLSA) est en $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$ (voir proposition 19 en annexe page 36). En outre, la ligne 11 incrémente la variable j au plus $|\mathcal{D}|$ fois ; la boucle ligne 9 s'exécute donc totalement en $\mathcal{O}(|\mathcal{D}|)$ (complexité amortie). □

L'algorithme 6 nous permet donc d'obtenir une PLSC de deux ordres à *buckets*, en un meilleur temps que la procédure décrite dans la section précédente, de la manière suivante :

1. Appels de l'algorithme 1 pour obtenir les ordres homogénéisés π_1^h et π_2^h :

$$\pi_2^h \leftarrow \text{HOMOGÉNÉISATION}(\pi_1, \pi_2)$$

$$\pi_1^h \leftarrow \text{HOMOGÉNÉISATION}(\pi_2, \pi_1)$$

2. Appel de l'algorithme 6 pour récupérer une PLSC τ à π_1 et π_2 :

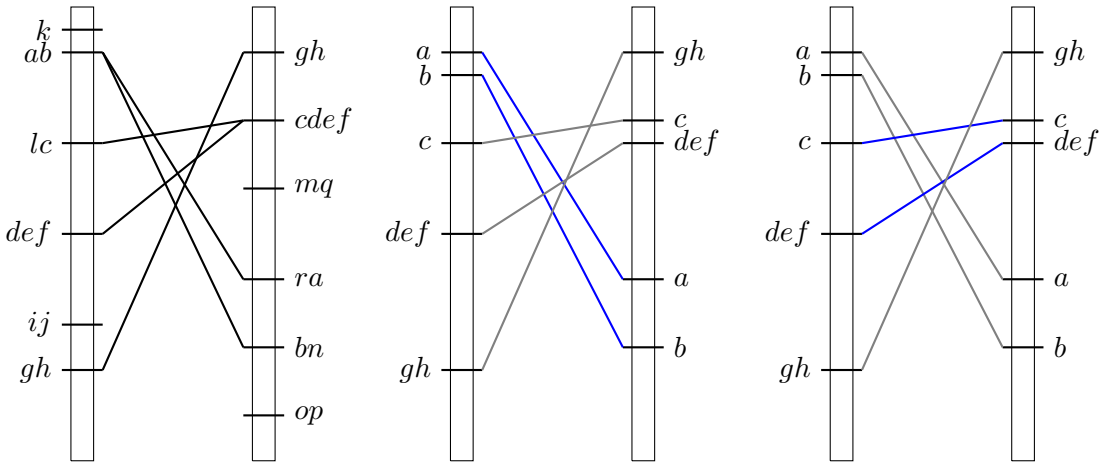
$$\tau \leftarrow \text{PLSC (PAR PLSA)}(\pi_1^h, \pi_2^h)$$

Sans perte de généralité, disons que $|\mathcal{D}_1| \leq |\mathcal{D}_2|$. La complexité en temps de cette procédure est $\mathcal{O}(|\mathcal{D}_2|log(|\mathcal{D}_1|)+|\mathcal{D}_1|log(|\mathcal{D}_2|))$ (voir propositions 3 page 11 et 18 page 29).

Bien que nous ne l'ayons pas décrit précisément, cette procédure est adaptable au calcul d'une PLSCI s. Pour cela, il suffit, non plus de récupérer exactement un élément de $\mathcal{D}$ par élément de la PLSA lors de la construction de s, mais de récupérer exactement un élément de $\mathcal{D}$ par ensemble maximum d'éléments consécutifs de la PLSA correspondant au même *bucket* des ordres homogénéisés (*cf.* exemple 11 page 25).

Soient $\pi_1 = (\{k\}, \{a, b\}, \{l, c\}, \{d, e, f\}, \{i, j\}, \{g, h\})$ et
$\pi_2 = (\{g, \}, \{c, d, e, f\}, \{m, q\}, \{r, a\}, \{b, n\}, \{o, p\})$ deux ordres à *buckets*, ainsi que
$\pi_1^h = (\{a\}, \{b\}, \{c\}, \{d, e, f\}, \{g, h\})$ et
$\pi_2^h = (\{g, h\}, \{c\}, \{d, e, f\}, \{a\}, \{b\})$ leurs ordres homogénéisés.

La figure 4.1 illustre la comparaison des cartes génomiques issues de π_1 et π_2 (4.1a), ainsi que la comparaison des cartes génomiques issues de π_1^h et π_2^h contenant un premier (correspondant à une PLSCI) (4.1b) et un second (correspondant à une PLSC mais aussi à une PLSCI) (4.1c) consensus sans conflit en bleu. Dans cette figure, un consensus sans conflit peut être vu comme un ensemble d'arêtes qui ne se croisent pas.



(a) Avant homogénéisation

(b) Un 1er consensus sans conflit après homogénéisation

(c) Un 2e consensus sans conflit après homogénéisation

FIGURE 4.1 – Comparaison de deux cartes génomiques, avant (4.1a) et après (4.1b,4.1c) homogénéisation. (4.1b) propose un premier consensus sans conflit, qui correspond à une PLSCI (en bleu), et (4.1c) en propose un second, qui correspond à une PLSC mais aussi à une PLSCI (en bleu).

Algorithme 6 : PLSC (par PLSA)

Données : Deux ordres à *buckets* π_1^h et π_2^h non vides d'un même domaine $\mathcal{D}$ issus d'une homogénéisation.

Résultat : P, une liste représentant une PLSC à π_1^h et π_2^h.

Soit $ext1posExt2$ un tableau vide de taille $|\mathcal{D}|$ de couples d'éléments (e, pos), $e \in \mathcal{D}$, $pos \in \mathbb{N}$.

$ext1posExt2 \leftarrow$ Codage Extensions Linéaires(π_1^h, π_2^h).

Soit tab un tableau vide de taille $|\mathcal{D}|$ d'entiers.

pour i *de* 1 *à* $|\mathcal{D}|$ **faire** $tab[i] \leftarrow ext1posExt2[i][2]$;

Soit L une liste d'entiers vide.

$L \leftarrow$ PLSA(tab) ;

Soit P une liste d'éléments de $\mathcal{D}$ vide.

$j \leftarrow 0$;

pour i *de* 1 *à* $|L|$ **faire**

    $j++$;

    tant que $L[i] \neq ext1posExt2[j][2]$ **faire** $j++$;

    P.push_back$(ext1posExt2[j][1])$;

retourner P ;

5

Conclusion et perspectives

5.1 Conclusion

Dans ce mémoire, nous avons proposé une formalisation d'une carte génomique en ordre à *buckets*. Un tel ordre permet de bien capturer le type d'informations contenu dans une carte : un *bucket* signale un ensemble de marqueurs dont l'ordre relatif est incertain sur un chromosome. À notre connaissance, une modélisation par les ordres à *buckets* des problèmes de génomique comparative n'a jamais été proposée dans la littérature. Une solution largement répandue pour représenter une carte génomique est l'utilisation de DAG[1] (*i.e.*, de graphes orientés acycliques) ; les ordres à *buckets* étant des structures plus simples à manipuler, leur utilisation est motivée par des temps de calcul potentiellement plus intéressants, tout en conservant une certaine souplesse pour la modélisation des imprécisions.

De plus, nous avons proposé un pré-traitement à réaliser sur deux ordres à *buckets* permettant de simplifier le calcul de leur consensus. Nous avons appelé ce pré-traimement une "homogénéisation", et exposé un premier algorithme, l'algorithme 1, qui homogénéise un ordre à *buckets* en temps logarithmique. Par la suite, nous avons adapté des idées provenant de [1] pour abaisser la complexité de l'homogénéisation à un temps linéaire. Cette amélioration de complexité est réalisée grâce à une nouvelle structure qui gère les éléments comme des pointeurs vers leurs valeurs.

Ce pré-traitement nous a permis de faciliter le calcul de trois variantes de consensus sans conflit présentées dans ce rapport : les PLSC et les PLSCI (chapitre 4), ainsi que l'ordre induit de deux ordres à *buckets* (annexe B). En effet, les corollaires 1 et 2 garantissent respectivement que les PLSC et les PLSCI à deux ordres à *buckets* sont exactement les mêmes que celles de leurs ordres homogénéisés, et le théorème 3 garantit que l'ordre induit de deux ordres à *buckets* est le même que celui de leurs ordres homogénéisés.

Concernant les problèmes de PLSC(I) à deux ordres à *buckets*, nous avons défini une PLSC de deux façons, dont l'une est plus stricte (PLSCI, qui impose que deux éléments consécutifs de la sous-séquence soient dans l'ordre croissant selon au moins un des deux ordres) que l'autre (PLSC, qui impose seulement que deux éléments consécutifs de la sous-séquence ne soient pas dans l'ordre décroissant selon au moins un des deux ordres).

[1] de l'anglais *directed acyclic graph*

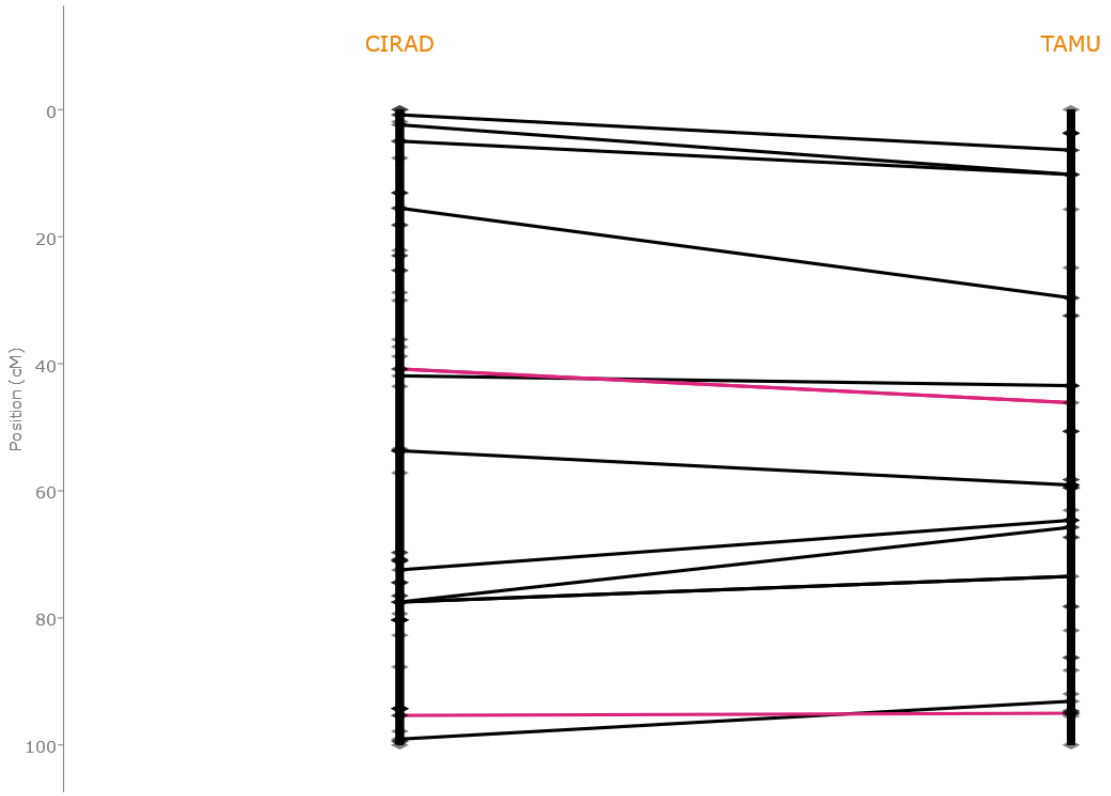


FIGURE 5.1 – Capture d'écran de l'outil *Genetic Map Comparator*

Nous avons proposé deux résolutions de ces deux variantes de consensus sans conflit : une première par un algorithme de programmation dynamique, et une deuxième par le calcul d'une PLSA. L'ordre induit à deux ordres à *buckets* peut être vu comme une généraliation d'une PLSCI. Nous avons adopté la même démarche et obtenu des résultats du même ordre que pour les problèmes de PLSC(I).

Par ailleurs, une première application pratique du travail effectué lors de ce stage est illustrée dans la figure 5.1. Cette figure contient une capture d'écran de l'outil *Genetic Map Comparator*[2], une application permettant de comparer des cartes génomiques. Ce stage permet de faire apparaître une PLSC s à deux cartes : elle correspond aux arêtes noires, qui relient deux mêmes éléments de s. Les marqueurs n'appartenant pas à la PLSC sont eux reliés par des arêtes roses.

[2]Disponible à l'adresse http://www.agap-sunshine.inra.fr/genmapcomp/

5.2 Perspectives

Le problème du consensus d'ordres à *buckets* comporte de nombreuses perspectives. Aussi, ce stage n'étant pas terminé, nous souhaitons investir les points suivants lors du mois à venir :

- implémentation des différents algorithmes proposés dans ce mémoire,
- s'intéresser à l'*informativité* d'un consensus (défini ci-dessous), pour pouvoir en comparer plusieurs en fonction des informations qu'ils contiennent.

  Soit σ un ordre sur un domaine $\mathcal{D}$. L'informativité de σ, noté $inf(\sigma)$, est donnée par la formule :

  $$inf(\sigma) = -log\left(\frac{|L(\sigma)|}{|\mathcal{D}|!}\right).$$

  Cette notion d'informativité [10] permet notamment de comparer la quantité d'informations contenues dans deux (ou plus) ordres σ_1 et σ_2 d'un même domaine $\mathcal{D}$. Pour cela, il suffit de remarquer que l'informativité de σ_1 est plus petite que celle de σ_2 si, et seulement si, le nombre d'extensions linéaires de σ_1 est supérieur à celui de σ_2, *i.e.*, $inf(\sigma_1) \leq inf(\sigma_2) \Leftrightarrow |L(\sigma_1)| \geq |L(\sigma_2)|$.

Par ailleurs, la modélisation d'une carte génomique par un ordre à *buckets* étant, à notre connaissance, nouvelle, il serait intéressant de revisiter différentes variantes du consensus de cartes génomiques sous ce type d'ordres (*e.g.* généraliser les PLSC(I) ou l'ordre induit (les résultats en annexe B page 38) à plus de deux ordres à *buckets*, ou encore au contraire ne plus se limiter à des consensus sans conflit, *e.g.* utiliser des méthodes de distances).

Annexes

A

Algorithme 7 : PLSA

Nous rappelons l'algorithme [9, 8] qui retourne, en temps $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$, une PLSA d'une suite d'entiers de taille $|\mathcal{D}|$.

Proposition 19. *L'algorithme 7 nécessite $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$ en temps.*

Démonstration. La boucle **"pour"** de la ligne 2 s'exécute $|\mathcal{D}|$ fois.

Aussi, la boucle **"pour"** de la ligne 5 s'exécute $|\mathcal{D}|$ fois. À chaque itération i de cette dernière, la boucle **"tant que"** de la ligne 13 réalise une recherche dichotomique dans $tab[1 \dots i]$. Une telle recherche pouvant s'implémenter en temps logarithmique en la taille de la structure, la boucle ligne 5 s'exécute totalement en $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$.

De plus, la boucle 25 récupère, à chaque itération, un élément de la PLSA en construction : elle s'exécute donc au plus $|\mathcal{D}|$ fois. Finalement, la complexité totale de l'algorithme 7 est $\mathcal{O}(|\mathcal{D}|log(|\mathcal{D}|))$. □

Algorithme 7 : PLSA

Données : Un tableau tab de taille $|\mathcal{D}| \leq 1$ d'éléments appartenant à un ensemble totalement ordonné.

Résultat : L, une liste représentant une PLSA de la suite induite par les entiers de $tab[1]$ à $tab[|\mathcal{D}|]$.

```
1  Soient T et R deux tableaux d'entiers de taille |D|.
2  pour i de 1 à |D| faire R[i] ← −1 ;
3  lastT ← 1 ;
4  T[lastT] ← 1 ;
5  pour i de 2 à |D| faire
6      si tab[i] > tab[T[lastT]] alors
7          T[lastT + 1] ← i ;
8          R[i] ← T[lastT] ;
9          lastT + + ;
10     sinon
11         lo ← 1 ;
12         hi ← lastT ;
13         tant que lo ≤ hi faire
14             mid ← ⌊(lo+hi)/2⌋ ;
15             si tab[T[mid]] < tab[i] alors
16                 lo ← mid + 1 ;
17             sinon
18                 hi ← mid − 1 ;
19         T[lo] ← i ;
20         si lo − 1 ≥ 1 alors R[i] ← T[lo − 1] ;
21 Soit L une liste d'entiers vide.
22 prev ← lastT ;
23 L.push_front(tab[prev]) ;
24 prev ← R[prev] ;
25 tant que prev ≠ −1 faire
26     L.push_front(tab[prev]) ;
27     prev ← R[prev] ;
28 retourner L ;
```

B

Ordre induit de deux ordres à *buckets*

Dans cette annexe, nous présentons un autre résultat de consensus sans conflit, qui généralise l'idée des PLSCI. On cherche ici l'ordre partiel qui inclut toutes les informations d'ordres présentes dans l'un ou l'autre des ordres d'entrée, et qui ne soit pas contredites par l'un ou l'autre de ces ordres. On nomme cet ordre *ordre induit*, et pour deux ordres donnés, il est défini de manière unique comme suit.

Définition 15 (Ordre induit de deux ordres)**.** *Soient π_1 et π_2 deux ordres à buckets de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$. L'ordre induit de π_1 et π_2 est l'ordre partiel σ_c sur $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$ qui satisfait, $\forall e_1, e_2 \in \mathcal{D}$,*

$$\begin{aligned} e_1 \prec_{\sigma_c} e_2 \Leftrightarrow & (e_1 \prec_{\pi_1} e_2 \text{ et } e_1 \prec_{\pi_2} e_2) \\ & \text{ou } (e_1 \prec_{\pi_1} e_2 \text{ et } e_1 \not\succ_{\pi_2} e_2) \\ & \text{ou } (e_1 \not\succ_{\pi_1} e_2 \text{ et } e_1 \prec_{\pi_2} e_2). \end{aligned}$$

Comme pour les PLSC(I), l'homogénéisation des deux ordres va nous permettre de faciliter les calculs tout en conservant l'équivalence des solutions, comme nous le montre le théorème suivant.

Théorème 3. *L'ordre induit σ_c de deux ordres à buckets π_1 et π_2 de domaines respectifs $\mathcal{D}_1$ et $\mathcal{D}_2$ est le même que l'ordre induit σ_c^h des ordres à buckets π_1^h et π_2^h issus respectivement de l'homogénéisation de π_1 et π_2.*

Démonstration. Pour cela, montrons les deux points suivant :

- **Montrons que les deux ordres induits σ_c et σ_c^h sont sur le même domaine $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$.**

  σ_c l'est par définition, et σ_c^h aussi puisque son domaine est l'intersection des domaines de π_1^h et π_2^h, *i.e.*, $\mathcal{D}$.

- **Montrons que $\sigma_c = \sigma_c^h$, *i.e.*, $\forall e_1, e_2 \in \mathcal{D}$, $e_1 \prec_{\sigma_c} e_2 \Leftrightarrow e_1 \prec_{\sigma_c^h} e_2$**

  $\Rightarrow e_1 \prec_{\sigma_c} e_2 \Rightarrow ((e_1 \prec_{\pi_1} e_2$ et $e_1 \prec_{\pi_2} e_2)$ ou $(e_1 \prec_{\pi_1} e_2$ et $e_1 \not\succ_{\pi_2} e_2)$ ou $(e_1 \not\succ_{\pi_1} e_2$ et $e_1 \prec_{\pi_2} e_2))$. Dans ces trois situations, par définition d'homogénéisation, le résultat $e_1 \prec_{\pi_1^h} e_2$ et $e_1 \prec_{\pi_2^h} e_2$ est toujours vérifié. Du coup, $e_1 \prec_{\sigma_c^h} e_2$.

$\Leftarrow e_1 \prec_{\sigma_c^h} e_2 \Rightarrow ((e_1 \prec_{\pi_1^h} e_2$ et $e_1 \prec_{\pi_2^h} e_2)$ ou $(e_1 \prec_{\pi_1^h} e_2$ et $e_1 \not\succ_{\pi_2^h} e_2)$ ou $(e_1 \not\succ_{\pi_1^h} e_2$ et $e_1 \not\prec_{\pi_2^h} e_2))$. Cependant, π_1^h et π_2^h étant issus d'une homogénéisation, ils contiennent les mêmes *buckets* : les deux derniers cas sont donc impossibles (e_1 et e_2 ne peuvent pas être à la fois comparables dans un ordre et incomparables dans l'autre). En outre, par définition d'homogénéisation, $(e_1 \prec_{\pi_1^h} e_2$ et $e_1 \prec_{\pi_2^h} e_2) \Rightarrow ((e_1 \prec_{\pi_1} e_2$ et $e_1 \prec_{\pi_2} e_2)$ ou $(e_1 \prec_{\pi_1} e_2$ et $e_1 \not\succ_{\pi_2} e_2)$ ou $(e_1 \not\succ_{\pi_1} e_2$ et $e_1 \prec_{\pi_2} e_2))$. Du coup, $e_1 \prec_{\sigma_c} e_2$.

□

L'algorithme 8 page 40 calcule l'ordre induit de deux ordres à *buckets* homogénéisés, en construisant un graphe G représentant l'ordre induit. Pour cela, l'algorithme commence par stocker dans les cases i d'un tableau T_1 le numéro du *bucket* de π_2^h qui contient le i^{e} *bucket* de π_1^h. Ensuite, l'algorithme parcourt les *buckets* B et B' de π_1^h qui sont tels que B est avant B' dans π_1^h, et grâce à T_1, on sait en temps constant s'ils sont aussi dans cet ordre dans π_2^h (auquel cas on ajoute l'arc correspondant dans G) ou non. Notons que le pré-traitement d'homogénéisation permet, comme pour les PLSC(I), de faciliter la résolution de cette variante de consensus sans conflit (théorème 3).

Proposition 20 (Terminaison de l'algorithme 8)**.** *L'algorithme 8 termine.*

Démonstration. Les boucles **"pour"** des lignes 2 à 6 itèrent sur des éléments finis. □

Proposition 21 (Correction de l'algorithme 8)**.** *L'algorithme 8 retourne le graphe orienté $G = (V, A)$ qui est tel que :*

- *$V = \{1, \ldots, b\}$, avec b le nombre de buckets de π_1^h,*
- *pour tous buckets B_i^1 et B_j^1 de π_1^h (représentés respectivement par les éléments i et j de V), pour tous les éléments e de B_i^1 et les éléments e' de B_j^1, $e \prec_{\sigma_c} e'$ si, et seulement si, $\overrightarrow{ij} \in A$, avec σ_c l'ordre induit de π_1^h et π_2^h.*

Démonstration. Montrons les deux points suivants :

- **G contient $|\mathcal{B}^1|$ sommets, et à chaque *bucket* de π_1^h et π_2^h est associé exactement un de ces sommets.**

  Le graphe $G = (V, A)$ est initialisé à vide. Les seuls ajouts dans V sont réalisés par la ligne 2 ; seuls les entiers 1 à $|\mathcal{B}^1|$ sont insérés dans V. Par convention, on note i le sommet de G associé au *bucket* B_i^1 de π_1^h (sachant que chacun de ces *buckets* appartient aussi à π_2^h). De plus, lors de l'ajout d'un arc $\overrightarrow{ij}$ dans G par l'intermédiaire de la ligne 8, les sommets i et j sont associés aux *buckets* B_i^1 et B_j^1 respectivement : les *buckets* sont identifiés par leurs positions dans π_1^h (par convention).

Algorithme 8 : GRAPHE DE L'ORDRE INDUIT

Données : π_1^h et π_2^h, deux ordres à *buckets* non vides d'un même domaine $\mathcal{D}$ issus d'une homogénéisation, dont les *buckets* sont ordonnés selon un ordre total sur $\mathcal{D}$.

Résultat : $G = (V, A)$, un graphe orienté tel que :

- $V = \{1, \ldots, b\}$, avec b le nombre de *buckets* de π_1^h et π_2^h,
- pour tous *buckets* B_i^1 et B_j^1 de π_1^h (représentés respectivement par les éléments i et j de V), pour tous les éléments e de B_i^1 et les éléments e' de B_j^1, $e \prec_{\sigma_c} e'$ si, et seulement si, $\overrightarrow{ij} \in A$, avec σ_c l'ordre induit de π_1^h et π_2^h.

Soient $\mathcal{B}^1 = (B_1^1, \ldots, B_{|\mathcal{B}^1|}^1)$ et $\mathcal{B}^2 = (B_1^2, \ldots, B_{|\mathcal{B}^2|}^2)$ les suites respectives des *buckets* de π_1^h et π_2^h, $G = (V, A)$ un multigraphe orienté vide, dic_2 un dictionnaire de hashage vide, ainsi que T_1 un tableau de $|\mathcal{B}^1|$ entiers vide.

```
// Création des sommets
```

pour i *de* 1 *à* $|\mathcal{B}^1|$ **faire** V.push_back(i);

pour i *de* 1 *à* $|\mathcal{B}^2|$ **faire** $dic_2.insert(key = B_i^2[1], value = i)$;

```
// Remplissage de T_1 avec les positions des buckets B_i^1 dans B^2
```

pour i *de* 1 *à* $|\mathcal{B}^1|$ **faire** $T_1[i] \leftarrow dic_2.getValue(B_i^1[1])$;

pour i *de* 1 *à* $|\mathcal{B}^1| - 1$ **faire**

  pour j *de* $i + 1$ *à* $|\mathcal{B}^1|$ **faire**

    si $T_1[i] < T_1[j]$ **alors**

      $A \leftarrow A \cup \{\overrightarrow{ij}\}$;

retourner G ;

- **Pour tous *buckets* B_i^1 et B_j^1 de π_1^h, l'arc $\overrightarrow{ij}$ est présent dans G si, et seulement si, $B_i^1 \prec_{\pi_1^h} B_j^1$ et $B_i^1 \prec_{\pi_2^h} B_j^1$.**

  Pour montrer ce point, commençons par prouver le lemme suivant.

  Lemme 3. *$T_1[i]$ contient la position du bucket de π_2^h qui est égal au i^e bucket de π_1^h, $\forall i \in \{1, \ldots, |\mathcal{B}^1|\}$.*

  $\triangle$ La ligne 3 insère, pour tout *bucket* B_i^2 de π_2^h, le premier élément de B_i^2 en tant que clé et la position i du *bucket* B_i^2 dans π_2^h dans le dictionnaire de hashage dic_2.

  La ligne 4 parcourt ensuite les *buckets* B_i^1 de π_1^h, de B_1^1 à $B_{|\mathcal{B}^1|}^1$: pour chacun de ces B_i^1, cette même ligne cherche la position de ce *bucket* dans π_2^h. Pour cela, il suffit de chercher dans dic_2 la valeur val associé à la clé $B_i^1[1]$. Comme cette valeur val correspond à la position du *bucket* B_i^1 dans π_2^h (explications du paragraphe précédent), le lemme est démontré. $\triangle$

  $\Leftarrow$ Si $B_i^1 \prec_{\pi_1^h} B_j^1$, alors il existe une itération de la boucle ligne 5 qui traite le *bucket* B_i^1 en même temps qu'une itération de ligne 6 traite le *bucket* B_j^1. En outre, comme $B_i^1 \prec_{\pi_2^h} B_j^1$, la condition "$T_1[i] < T_1[j]$" ligne 7 est satisfaite (voir lemme 3). L'arc $\overrightarrow{ij}$ est alors ajouté dans A (ligne 8).

  $\Rightarrow$ Si un arc $\overrightarrow{ij}$ est créé, c'est que la condition de la ligne 7 est remplie pour les *buckets* B_i^1 et B_j^1. Par le lemme 3, $B_i^1 \prec_{\pi_2^h} B_j^1$. Aussi, comme $j > i$, $B_i^1 \prec_{\pi_1^h} B_j^1$.

- **Pour tous les éléments e et e' de $\mathcal{D}$, $e \prec_{\sigma_c} e'$ si, et seulement si, l'arc $\overrightarrow{ij}$ est présent dans G, avec i la position du *bucket* de π_1^h contenant e et j celle du *bucket* de π_1^h contenant e'.**

  Par montrer cela, nous commençons par prouver la propriété suivante :

  Propriété 5. *Soient π_1^h et π_2^h deux ordres à buckets issus d'une homogénéisation, ainsi que B_i et B_j des buckets de π_1^h et π_2^h. L'ordre induit σ_c de π_1^h et π_2^h est tel que $\forall e \in B_i$, $\forall e' \in B_j$, $B_i \prec_{\pi_1^h} B_j$ et $B_i \prec_{\pi_2^h} B_j$ si, et seulement si, $e \prec_{\sigma_c} e'$.*

  $\triangle$ Par définition, l'ordre induit σ_c de deux ordres à *buckets* π_1^h et π_2^h est tel que $\forall e_1, e_2 \in \mathcal{D}$,

$$\begin{aligned} e_1 \prec_{\sigma_c} e_2 \Leftrightarrow & (e_1 \prec_{\pi_1^h} e_2 \text{ et } e_1 \prec_{\pi_2^h} e_2) \\ & \text{ou } (e_1 \prec_{\pi_1^h} e_2 \text{ et } e_1 \not\prec_{\pi_2^h} e_2) \\ & \text{ou } (e_1 \not\prec_{\pi_1^h} e_2 \text{ et } e_1 \prec_{\pi_2^h} e_2). \end{aligned}$$

  Comme π_1^h et π_2^h sont issus d'une homogénéisation, ils contiennent les mêmes *buckets* mais pas forcément dans le même ordre. Du coup, les cas "$e_1 \prec_{\pi_1^h} e_2$ et $e_1 \not\prec_{\pi_2^h} e_2$" et "$e_1 \not\prec_{\pi_1^h} e_2$ et $e_1 \prec_{\pi_2^h} e_2$" sont irréalisables.

Il en reste donc que $e_1 \prec_{\sigma_c} e_2$ si, et seulement si, $e_1 \prec_{\pi_1^h} e_2$ et $e_1 \prec_{\pi_2^h} e_2$. Soient B_1 le *bucket* contenant e_1 et B_2 celui contenant e_2. Comme π_1^h et π_2^h contiennent les mêmes *buckets*, $e_1 \prec_{\pi_1^h} e_2$ et $e_1 \prec_{\pi_2^h} e_2$ si, et seulement si, $B_1 \prec_{\pi_1^h} B_2$ et $B_1 \prec_{\pi_2^h} B_2$. Finalement, $e_1 \prec_{\sigma_c} e_2$ si, et seulement si, $B_1 \prec_{\pi_1^h} B_2$ et $B_1 \prec_{\pi_2^h} B_2$. △

Cette propriété ainsi que le point précédent nous permettent de montrer le résultat attendu.

□

Proposition 22 (Complexité en temps de l'algorithme 8)**.** *L'algorithme 8 nécessite* $\mathcal{O}(|\mathcal{B}^1|^2)$ *en temps, avec* $|\mathcal{B}^1|$ *le nombre de buckets des ordres d'entrée.*

Démonstration. La boucle **"pour"** ligne 2 s'exécute $|\mathcal{B}^1|$ fois. Les boucles **"pour"** des lignes 3 et 4 s'exécutent chacune $|\mathcal{B}^1|$ fois, et les instructions qu'elles contiennent sont en $\mathcal{O}(log(|\mathcal{B}^1|))$ (insertion ou récupération d'une valeur dans un dictionnaire de hashage) : la complexité totale de chacune de ces deux boucles est donc $\mathcal{O}(|\mathcal{B}^1|log(|\mathcal{B}^1|))$. Les boucles **"pour"** lignes 5 et 6 s'exécutent au plus $|\mathcal{B}^1|$ fois chacune : elles sont donc en $\mathcal{O}(|\mathcal{B}^1|^2)$.
□

Bibliographie

[1] Mukul S. Bansal et David Fernández-Baca : Computing distances between partial rankings. *Inf. Process. Lett.*, 109(4):238–241, 2009.

[2] Franz J. Brandenburg et Andreas Gleissner : Ranking chain sum orders. *Theor. Comput. Sci.*, 636:66–76, 2016. URL https://doi.org/10.1016/j.tcs.2016.05.026.

[3] Franz J Brandenburg, Andreas Gleissner et Andreas Hofmeier : The nearest neighbor Spearman footrule distance for bucket, interval, and partial orders. *In Frontiers in Algorithmics and Algorithmic Aspects in Information and Management*, pages 352–363. Springer, 2011.

[4] Franz-Josef Brandenburg, Andreas Gleissner et Andreas Hofmeier : Comparing and Aggregating Partial Orders with Kendall tau Distances. *Discrete Math., Alg. and Appl.*, 5(2), 2013.

[5] Franz-Josef Brandenburg, Andreas Gleissner et Andreas Hofmeier : The nearest neighbor spearman footrule distance for bucket, interval, and partial orders. *J. Comb. Optim.*, 26(2):310–332, 2013. URL https://doi.org/10.1007/s10878-012-9467-x.

[6] Jeffrey B Endelman : New algorithm improves fine structure of the barley consensus SNP map. *BMC Genomics*, 12(1):407, 2011.

[7] Jeffrey B Endelman et Christophe Plomion : LPmerge : an R package for merging genetic maps by linear programming. *Bioinformatics*, 30(11):1623–1624, 2014.

[8] Donald E Knuth : Sorting and searching. 1973.

[9] C Schensted : Longest increasing and decreasing subsequences. *order*, 7:59, 1961.

[10] Celine Scornavacca, Vincent Berry, Vincent Lefort, Emmanuel JP Douzery et Vincent Ranwez : Physic_ist : cleaning source trees to infer more informative supertrees. *BMC bioinformatics*, 9(1):413, 2008.

[11] Haibao Tang, Xingtan Zhang, Chenyong Miao, Jisen Zhang, Ray Ming, James C Schnable, Patrick S Schnable, Eric Lyons et Jianguo Lu : ALLMAPS : robust scaffold ordering based on multiple maps. *Genome Biology*, 16(1):3, 2015.

[12] Robert A Wagner et Michael J Fischer : The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[13] Immanuel V. Yap, David Schneider, Jon Kleinberg, David Matthews, Samuel Cartinhour et Susan R. McCouch : A Graph-Theoretic Approach to Comparing and Integrating Genetic, Physical and Sequence-Based Maps. *Genetics*, 165(4): 2235–2247, 2003.