UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UNE NOUVELLE APPROCHE COMPUTATIONNELLE POUR LA GÉNÉRATION DE GRAPHES DE RECOMBINAISON ANCESTRAUX COMPORTANT UN GRAND NOMBRE DE MARQUEURS GÉNÉTIQUES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

ÉRIC MARCOTTE

SEPTEMBRE 2018

REMERCIEMENTS

Tout d'abord, j'aimerais remercier ma mère qui m'a toujours impressionné par l'étendue de ses connaissances du monde de la philosophie. Il n'en fallait pas plus pour que je sois en mesure d'apprécier et de valoriser le travail intellectuel. Grâce à toi, aujourd'hui, ma soif pour les sciences guide mon parcours.

J'aimerais aussi remercier Fabrice qui m'a orienté avec patience et sagesse dans mon travail d'exploration de ce sujet intéressant. Merci de m'avoir donné l'opportunité de découvrir comment les mathématiques pouvaient s'arrimer avec tant d'élégance et de justesse au monde naturel.

À mes enfants, Maëva et Tristan.

TABLE DES MATIÈRES

LISTE DES TABLEAUX xi
LISTE DES FIGURES xiii
RÉSUMÉ xix
INTRODUCTION 1
CHAPITRE I
INTRODUCTION À LA GÉNÉTIQUE 5
1.1 L'ADN 5
1.1.1 L'ADN chez l'humain 6
1.1.2 Binarisation des séquences 7
1.2 Évolution de l'ADN 8
1.2.1 Les mutations 8
1.2.2 Les recombinaisons 10
1.3 Les liaisons génétiques 11
1.3.1 Le déséquilibre de liaison 13
CHAPITRE II
LES GRAPHES PHYLOGÉNÉTIQUES 17
2.1 Phylogénie parfaite 17
2.2 Le graphe de recombinaison ancestral 21
2.2.1 Présentation générale 21
2.2.2 Les opérations 23
2.2.3 Algorithme de construction 25
2.3 Utilité des graphes 26
CHAPITRE III
STRATÉGIE COMPUTATIONNELLE 29

3.1 Approche ensembliste 29
3.1.1 Fonctionnement des coalescences 29
3.1.2 Fonctionnement des mutations et des recombinaisons 31
3.1.3 Avantages et désavantages de cette approche 32
3.2 Approche par fonction booléenne 32
3.2.1 Encodage des séquences 33
3.2.2 Fonctionnement des coalescences 34
3.2.3 Fonctionnement des mutations et des recombinaisons 41
3.2.4 Avantages et désavantages de cette approche 41
CHAPITRE IV
DÉRIVE GÉNÉTIQUE 43
4.1 Le modèle Wright–Fisher 43
4.2 Théorie de la coalescence 47
4.2.1 Théorie de la coalescence avec mutations 50
CHAPITRE V
ALGORITHMES DE CONSTRUCTION 53
5.1 Notations des opérations 54
5.1.1 Les coalescences 54
5.1.2 Les mutations 54
5.1.3 Les recombinaisons 55
5.2 Caractéristiques des ARGs 56
5.3 Heuristiques 57
5.3.1 Margarita 57
5.3.2 ARG4WG 62
5.3.3 Manhattan 64
5.4 Distribution statistique 68
5.4.1 Distribution optimale 68
5.4.2 Fonction de densité 71

5.4.3 Densité alternative 74

CHAPITRE VI
IMPLÉMENTATION 77

6.1 Type abstrait de données 77

6.2 Fonctionnement général de *FastARG* 79

6.2.1 Implémentation du test de coalescence 80

6.2.2 Test de coalescence avec contrainte 85

6.2.3 Calcul des coalescences 86

6.2.4 Implémentation des mutations et des recombinaisons 90

6.2.5 Extraction des topologie 94

6.2.6 Implémentation d'algorithme générale 96

6.3 Les statistiques 97

6.3.1 Test du χ^2 97

6.3.2 Test par percolation 98

6.3.3 Constructions de nouveaux graphes 99

6.4 Résultats 101

CONCLUSION 105

ANNEXE A
DÉMONSTRATIONS SUPPLÉMENTAIRES DE LA FONCTION TESTANT L'INCOMPATIBILITÉ D'UN LOCUS. 107

ANNEXE B
IMPLÉMENTATION DE L'ALGORITHME DE MARGARITA 109

ANNEXE C
IMPLÉMENTATION DES TESTS UNITAIRES 111

ANNEXE D
CODE MAITRE DE LA FONCTION PARALLÈLE SUR GPU 115

RÉFÉRENCES 121

LISTE DES TABLEAUX

Tableau		Page
1.1	Illustration de la variation génétique sur cinq séquences.	7
1.2	Illustration de l'encodage des SNPs	8
1.3	Probabilités pour deux *loci*.	14
2.1	Données pour la phylogénie parfaite	19
3.1	Combinaisons possibles de paires de *loci*	30
3.2	Équivalence entre les symboles logiques et les opérations bit à bit correspondantes en *C/C++*.	33
3.3	Table d'équivalence des états possibles	34
3.4	Table de vérité pour une coalescence	35
3.5	Table de Karnaugh pour la coalescence	36
3.6	Table de vérité pour une coalescence stricte	40
4.1	Résumé des équations de la théorie de la coalescence	49
4.2	Temps moyen avant MRCA	50
6.1	Spécification sémantique de l'objet ARG	78
6.2	Test du χ^2	98
6.3	Résultats des tests sur 500 haplotypes	102
6.4	Résultats des tests sur 1000 haplotypes	103
6.5	Résultats des tests sur 2000 haplotypes	104

LISTE DES FIGURES

Figure		Page
0.1	Darwin et l'arbre de la vie.	2
1.1	Transtion v.s. transvertion	11
1.2	Illustration d'une recombinaison	12
1.3	Carte des gènes de Thomas Hunt Morgan	12
1.4	Illustration du déséquilibre de liaison	15
2.1	Flux générique de la construction d'une phylogénie.	18
2.2	Illustration du résultat de l'algorithme	21
2.3	Un graphe de recombinaison ancestral simple	22
2.4	Exemple d'une mutation	23
2.5	Exemple d'une coalescence	24
2.6	Exemple d'une recombinaison	24
5.1	Recombinaison de type Margarita	59
5.2	ARG construit selon l'algorithme Margarita	60
5.3	Recombinaison de type ARG4WG	63
5.4	Les recombinaisons selon Manhattan	65
5.5	ARG construit selon Margarita	67
6.1	Temps d'exécution en fonction du nombre de bits	83
6.2	Temps d'exécution en fonction de la taille des grains	84
6.3	Temps de coalescence en fonction du nombre de fils d'exécutions	87
6.4	Temps d'éxécution sur processeur graphique	90

6.5 Propagation de mutations dans un ARG. 100

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

ADN Acide désoxyribonucléique. 1, 2, 5, 6, 8, 9

ARG Pour «*Ancestral recombination graph*» en anglais, ou graphe de recombinaison ancestral. 3, 17, 21, 23–26, 41, 52, 53, 56, 57, 61, 62, 66, 68, 69, 71, 77–79, 87, 91, 92, 96, 97, 99, 101, 106

ARG4WG Algorithme de création d'ARG décrit dans l'article (Nguyen *et al.*, 2017).

ARN Acide ribonucléique. 8

JSON Pour «*JavaScript object notation*» en anglais.

LOD Pour «*Logarithm of odds*» en anglais.

Margarita Algorithme de création d'ARG décrit dans l'article (Minichiello et Durbin, 2006).

MRCA Pour « Most recent common ancestor » en anglais. 19, 21, 48–50, 60, 64, 66, 99, 101

SNP Pour «*Single-nucléotide polymophism*» en anglais. xi, 7, 8, 32, 58, 64, 79, 82, 84, 86, 87, 104

TBB Pour «*Threading building blocks*» en anglais.

TIM Pour «*Trait-inflencing mutation*» en anglais. 97, 99

LISTE DES SYMBOLES ET DES UNITÉS

$\&$ Opérateur binaire AND en *C++*.

Δ Différence symétrique entre deux ensembles.

■ Représentation d'un marqueur mutant.

□ Représentation d'un marqueur primitif.

⊠ Représentation d'un marqueur non-ancestral.

δ Symbole de Kronecker.

$\wedge$ Conjonction logique.

μ Taux de mutation.

$\neg$ Négation logique.

$\oplus$ Ou exclusif logique.

$\sim$ Opérateur unaire NOT en *C++*.

$\vee$ Disjonction logique.

$|$ Opérateur binaire OR en *C++*.

$\hat{}$ Opérateur binaire XOR (ou exclusif) en *C++*.

RÉSUMÉ

Nous présentons dans ce mémoire une nouvelle approche permettant de créer et d'exploiter des graphes de recombinaison ancestraux comportant un grand nombre de marqueurs génétiques. Nous utilisons principalement une représentation binaire ainsi que des fonctions booléennes applicables sur des séquences de marqueurs afin d'implémenter un logiciel fait sur mesure pour accomplir cette tâche de manière efficiente. De plus, nous détaillons une heuristique de construction bien connue et nous proposerons un algorithme novateur minimisant le nombre de recombinaisons requises pour parvenir à un ancêtre commun. Enfin, nous exposons quelques résultats sur l'efficacité de notre approche.

Mots clés : graphe de recombinaison ancestral, cartographie génétique, processus de coalescence, heuristiques, mutation.

INTRODUCTION

Depuis 1859, lors de la publication du célèbre ouvrage de Charles Darwin sur l'origine des espèces (Darwin, 1859), l'Homme a toujours rêvé de connaitre l'histoire complète de la vie. Le concept d'arbre de la vie au sens biologique a ainsi vu le jour. Cette théorie stipule que chacun d'entre nous possède un ancêtre commun avec tout autre organisme vivant sur la planète. Cette idée constitue une des plus grandes percées scientifiques accomplies par l'esprit humain. Puisque certains de nos prédécesseurs sont morts depuis maintenant plus de 4 milliards d'années, nous ne pourrons probablement jamais savoir exactement comment la vie a évolué, ni même de quelle manière elle est apparue. Mais qui reste indifférent à l'idée de pouvoir observer l'ancêtre commun que nous partageons avec cet arbre que nous avons escaladé lorsque nous étions enfant ?

Beaucoup de temps s'est écoulé depuis Darwin et la découverte de l'acide désoxyribonucléique, ou ADN. Nos connaissances continuent sans cesse de s'affiner et de confirmer sans aucun doute la théorie de l'évolution de Darwin. Aujourd'hui, ces notions sont mises à contribution pour peaufiner notre reconstitution de l'évolution de la vie. De façon plus générale, la phylogénie étudie les liens évolutifs entre les êtres vivants. Le niveau généalogique s'intéresse surtout à l'étude entre individus dans une population. Le niveau interspécifique quant à lui, explore les liens entre des espèces différentes et rend une analyse plus globale. Ces liens peuvent être intuitivement représentés au moyen d'un graphe au sens mathématique. Ce graphe prend souvent la forme d'un arbre, car il est plus facile de comprendre les

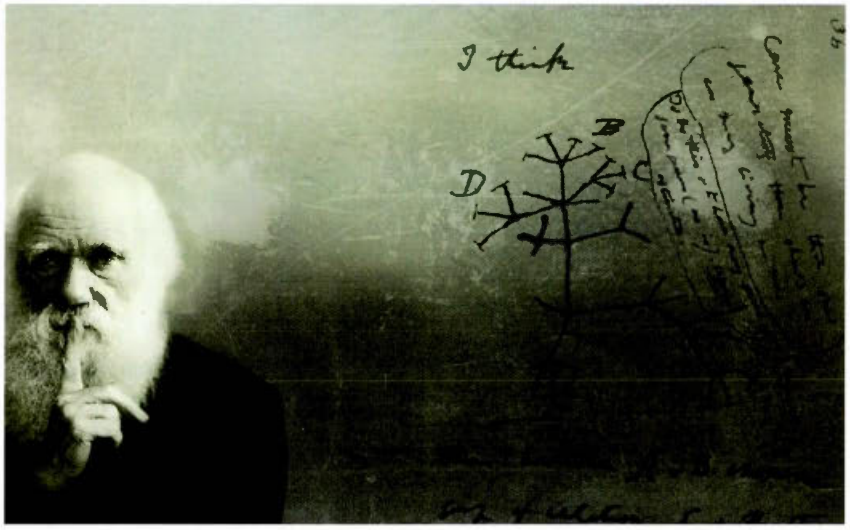


Figure 0.1 Darwin et un dessin provenant d'un cahier de notes conceptualisant un des premiers arbres évolutionnaires. [2].

relations de parenté pour un ensemble d'organismes donnés sous cette représentation. Un arbre généalogique est un exemple simpliste d'une phylogénie. Lors de la genèse de cette science, les arbres étaient surtout basés sur des observations qualitatives, comme la variation de la forme du bec de plusieurs espèces d'oiseaux par exemple. Le degré de certitude des inférences était alors relativement faible. La découverte et le séquençage de l'ADN en revanche ont permis d'augmenter énormément la qualité de ces arbres. La performance et le coût des séquenceurs s'étant grandement améliorés, la quantité d'informations génétiques disponibles dans les diverses banques de données a explosé depuis quelques années. Actuellement, on peut facilement obtenir les données nécessaires pour effectuer de l'inférence phylogénétique. Ces phylogénies sont très utiles dans plusieurs domaines et ont parfois des applications concrètes.

---

2. Source : http://curiouser.fr/conference-de-brian-solis-on-y-etait/darwin/

Dans les chapitres qui suivent, nous nous penchons surtout sur deux sortes particulières de graphes phylogénétiques, soit le graphe de recombinaison ancestral qui s'appuie sur le processus de coalescence lors de son inférence ainsi qu'un arbre phylogénétique interespèce. Plus spécifiquement, nous nous intéressons à la manière de générer ces graphes rapidement sur un grand nombre de marqueurs génétiques ainsi que certaines méthodes pour extraire un maximum d'information statistique.

Avant de nous engager plus en profondeur dans le graphe de recombinaison ancestral, nous introduisons certains concepts génétiques importants dans les deux chapitres suivants. Puis, nous détaillons les particularités des graphes de recombinaison ancestraux (ARGs) lors du troisième et du quatrième chapitre. Le chapitre cinq sera consacré aux différents algorithmes et méthodes statistiques appliquées aux ARGs. Enfin, nous concluons avec les résultats obtenus avec notre nouvelle approche.

CHAPITRE I

INTRODUCTION À LA GÉNÉTIQUE

La découverte de l'acide désoxyribonucléique ADN en 1869 par Friedrich Miescher, et sa structure en double hélice en 1953 par Francis Crick et James Watson, a permis de propulser la phylogénie à de nouveaux sommets. C'est cette molécule qui régit les principes fondamentaux de l'hérédité. Gregor Mendel est un des premiers scientifiques à avoir observé quelques spécificités de la transmission des caractères, et ce avant l'identification de l'ADN.

1.1 L'ADN

L'ADN est la molécule qui codifie en grande partie les traits d'un organisme. L'information est principalement contenue dans la disposition des deux chaines complémentaires polynucléotidiques. Ces quatre bases nucléiques sont l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T). L'adénine et la thymine forment une paire complémentaire, tandis que la guanine et la cytosine forment l'autre couple. Donc, il suffit d'obtenir l'information d'un seul côté d'un brin de l'ADN pour déduire la base nucléique sur l'autre brin. En biologie moléculaire, on parle de brin de polarité positive ou négative pour distinguer les deux séquences. Un autre concept important, la *ploïdie* d'un organisme, correspond au nombre d'exemplaires des chromosomes homologues. On parle alors d'organismes haploïdes pour ceux qui possèdent un seul jeu de chromosomes homologues. L'es-

pèce humaine possède deux jeux de chromosomes homologues ce qui fait d'elle un organisme diploïde. Certains organismes en détiennent trois (triploïdie) et même quatre (tétraploïdes) comme le saumon par exemple.

1.1.1 L'ADN chez l'humain

L'humain possède 23 paires de chromosomes qui contiennent environ 3,2 milliards de paires de bases, 6,4 milliards en comptant les chromosomes homologues. Nous possédons donc une partie de notre matériel génétique en double. Pour chaque couple de chromosomes homologue, un provient du côté maternel et un autre du côté paternel. Pour chaque position possible de l'ADN, on doit connaitre les deux bases qui constituent les «versions» que nous détenons. On parle alors des *allèles* pour chacune des «versions» et des *loci* (*locus* au singulier) pour les positions. L'haplotype d'un individu est déterminé lorsqu'on possède le génotype des chromosomes homologues. Bien sûr, comme l'ADN est constitué de deux brins, on compare toujours les allèles selon la même orientation, les deux brins avec la même polarité dans ce cas. Notons que ceci ne constitue pas la totalité du *génotype* d'une personne. En effet, d'autres sources existent, certaines sources sont triviales, comme l'ADN mitochondrial, mais d'autres sont surprenantes, comme l'existence de cellules mâles dans le cerveau de femmes provenant de fœtus qu'elles ont portés (Chan *et al.*, 2012). Toutefois d'un humain à un autre, la variation du génotype est très faible, de l'ordre de 1%. Un locus qui possède une variation allélique assez grande à l'intérieur d'une population peut faire office de marqueur génétique comme illustré dans le tableau 1.1. De plus, les *loci* avec un faible taux de mutation récurrente ont généralement seulement deux allèles possibles.

Tableau 1.1 Cinq séquences fictives de vingt bases nucléiques. Les positions : 2, 6, 8, 14, 17 *et* 20 *sont les seules à posséder des variations.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
s_1	T	C	T	T	A	G	G	T	T	G	A	A	T	C	C	G	G	A	A	T
s_2	T	A	T	T	A	A	G	T	T	G	A	A	T	C	C	G	G	A	A	T
s_3	T	C	T	T	A	A	G	A	T	G	A	A	T	T	C	G	G	A	A	T
s_4	T	C	T	T	A	A	G	T	T	G	A	A	T	T	C	G	G	A	A	G
s_5	T	C	T	T	A	G	G	T	T	G	A	A	T	C	C	G	C	A	A	T

1.1.2 Binarisation des séquences

Comme on peut le voir dans le tableau 1.1, une grande partie de l'information s'avère redondante. Lorsqu'on s'intéresse seulement aux liens phylogénétiques entre un groupe d'individus par exemple, il n'est pas essentiel de conserver toutes les données génétiques disponibles. De ce fait, on peut réduire substantiellement la quantité d'information nécessaire pour encoder les séquences. Pour y parvenir, on s'intéresse souvent à des sous-groupes de ces bases nucléiques. C'est avec ces bases polymorphiques que l'on surnomme «SNP» (pour «*single-nucleotide polymorphism*» en anglais) qu'on fonde les modèles d'inférence phylogénétique. La prospection et la caractérisation de ces positions sont toujours des domaines très actifs de recherche. En 2001, on dénombrait environ 1,42 million (Nature, 2001) de SNPs dans le génome humain. Puis en 2007, ce nombre était passé à 3,1 millions (Nature, 2007). En 2010, ce chiffre a encore augmenté à 15 millions (Nature, 2010). Actuellement selon le *NCBI*, il y aurait plus de 113 millions de SNPs validés et possiblement plus de 150 millions [1]. Le tableau 1.2 illustre la façon dont les séquences du tableau 1.1 sont encodées. L'encodage binaire est beaucoup plus efficient tout en capturant la majorité de la variation génétique.

1. http://www.ncbi.nlm.nih.gov/projects/SNP/snp_summary.cgi 10 août 2018

Tableau 1.2 Illustration de l'encodage des SNPs

	2	6	8	14	17	20
s_1	C	G	T	C	G	T
s_2	A	A	T	C	G	T
s_3	C	A	A	T	G	T
s_4	C	A	T	T	G	G
s_5	C	G	T	C	C	T

(a) Extraction des SNPs.

	2	6	8	14	17	20
s_1	0	0	0	0	0	0
s_2	1	1	0	0	0	0
s_3	0	1	1	1	0	0
s_4	0	1	0	1	0	1
s_5	0	0	0	0	1	0

(b) Représentation binaire.

1.2 Évolution de l'ADN

Plusieurs mécanismes permettent aux séquences et donc aux organismes de brasser leur matériel génétique. Pour mieux comprendre comment l'ADN a permit l'apparition d'une grande diversité d'organismes, nous nous intéressons, dans cette section, à seulement deux des dispositifs les plus importants : les mutations et les recombinaisons.

1.2.1 Les mutations

Le code génétique fluctue à travers le temps et plusieurs types de mutations avec des causes différentes peuvent survenir. Il existe plusieurs catégories contenant plusieurs types de mutations génétiques. Dans la catégorie des mutations que l'on pourrait qualifier de simple, nous retrouvons les mutations par substitution, par insertion et par délétion, par exemple. Comme il est possible qu'une seule mutation altère considérablement l'expression d'un gène, il est facile d'imaginer qu'elle pourrait provoquer une maladie chez le porteur. Même dans le cas d'une mutation silencieuse, où la substitution d'une base nucléique ne modifie pas la séquence d'acides aminés d'une protéine, elle pourrait entrainer de graves conséquences si elle donnait lieu à une erreur d'épissage (lors de la transcription de l'ADN en ARN messager) par exemple.

Le processus de mutagenèse constitue un des vecteurs responsables de l'évolution des espèces. On parle de microévolution lorsque l'étude d'un ensemble de séquences génétiques provient d'une seule espèce. Formellement, la microévolution est le changement de la fréquence allélique d'un locus à travers le temps. De cette manière, on peut définir des modèles d'évolution de l'ADN qui permettent, entre autres, d'obtenir une mesure de la distance entre deux séquences. Voici deux modèles d'évolution de l'ADN qui cherchent à quantifier une distance évolutionnaire entre deux séquences.

1.2.1.1 Modèle de Jukes-Cantor

Un des modèles les plus simples est celui de Jukes-Cantor (Jukes *et al.*, 1969) qui suppose que la fréquence allélique est identique ($\pi_A = \pi_G = \pi_C = \pi_T = 0.25$) pour les quatre bases. Le modèle suppose aussi que le taux de mutation μ est aussi identique pour toutes les mutations. La matrice des taux de mutation est :

$$Q = \begin{pmatrix} * & \frac{\mu}{4} & \frac{\mu}{4} & \frac{\mu}{4} \\ \frac{\mu}{4} & * & \frac{\mu}{4} & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4} & * & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4} & \frac{\mu}{4} & * \end{pmatrix}. \tag{1.1}$$

Les termes sur la diagonale dans la matrice Q dénotent le fait qu'une mutation qui remplace par la base d'origine n'est pas vraiment une mutation au sens génétique. De la matrice Q on obtient la matrice P :

$$P_{ij}(\nu) = \begin{cases} \frac{1}{4} + \frac{3}{4}e^{-4\nu/3} & \text{si } i = j \\ \frac{1}{4} - \frac{1}{4}e^{-4\nu/3} & \text{si } i \neq j \end{cases},$$

qui correspond aux probabilités de transition en fonction du temps. Notons que $\nu = \frac{3}{4}t\mu$, où ν est la distance mesurée entre deux séquences. Soit p la proportion de sites où deux séquences ont des allèles différents, alors l'estimation de la distance évolutionnaire $\hat{d}$ de Jukes-Cantor est donnée par l'équation suivante :

$$\hat{d} = -\frac{3}{4}\ln\left(1 - \frac{4}{3}p\right) = \hat{\nu}. \tag{1.2}$$

1.2.1.2 Modèle de Kimura

Le modèle de Kimura (Kimura, 1980) est plus riche, car il distingue deux types de mutations par substitution, soit les transitions et les transversions telles qu'illustrées dans la figure 1.1. D'autres types de mutation sont possibles, comme les délétions, les insertions et les duplications, mais ils ne sont pas pris en compte dans ce modèle. Comme il existe des mécanismes chimiques précis qui influent sur la probabilité de voir ces deux types de mutations par substitution, l'ajout de ce paramètre permet d'enrichir les modèles l'utilisant. Ce modèle suppose aussi que la fréquence allélique est uniforme, c'est-à-dire que $\pi_A = \pi_G = \pi_C = \pi_T = 0.25$. La distance de Kimura est donnée par l'équation suivante :

$$K = -\frac{1}{2}\ln\left((1 - 2p - q)\sqrt{1 - 2q}\right), \tag{1.3}$$

où p est la proportion des *loci* où une mutation de type transition a eu lieu et q est la proportion des transversions.

1.2.2 Les recombinaisons

Les recombinaisons permettent de transmettre un mélange des deux chromosomes homologues que possède un individu à sa descendance. La descendance peut alors hériter de certaines caractéristiques qui ne sont pas présentes chez les parents. Divers mécanismes naturels sont possibles généralement lors d'une méiose, mais aussi lors d'une mitose. Les virus peuvent aussi recombiner en mélangeant deux

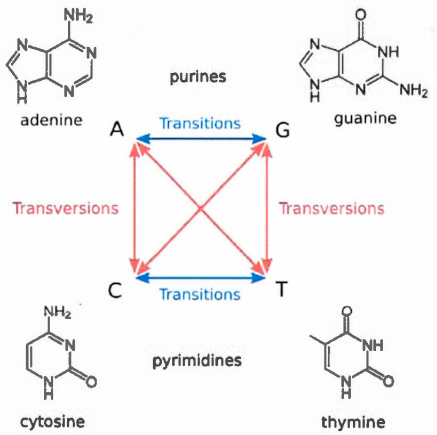


Figure 1.1 Illustration de la différence entre les mutations de transitions et les transversions. [1]

virus distincts. Comme le taux de recombinaison n'est pas uniforme sur tous les chromosomes, on peut réussir à inférer un génotype plus facilement. La figure 1.2 illustre le brassage génétique sur un chromosome lors d'une méiose.

1.3 Les liaisons génétiques

Les liaisons génétiques (*«linkage»* en anglais) expriment le fait que deux gènes ont une plus grande probabilité d'être transmis ensemble s'ils sont très proches physiquement l'un de l'autre. En effet, lors d'une méiose par exemple il y a moins de risque qu'un enjambement survienne entre deux *loci* proches. Un des exemples célèbres de l'utilisation des liaisons génétiques est la carte des gènes d'une mouche drosophile créée par Thomas Hunt Morgan illustrés à la figure 1.3. L'étude des maladies possédant une pénétrance forte ont généralement recours aux liaisons génétiques pour découvrir quel gène est responsable. On entend par pénétrance forte, le fait de posséder une certaine mutation qui rend très probable l'apparition d'un trait phénotypique, comme une maladie par exemple.

---

1. By Petulda (Own work) [CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons

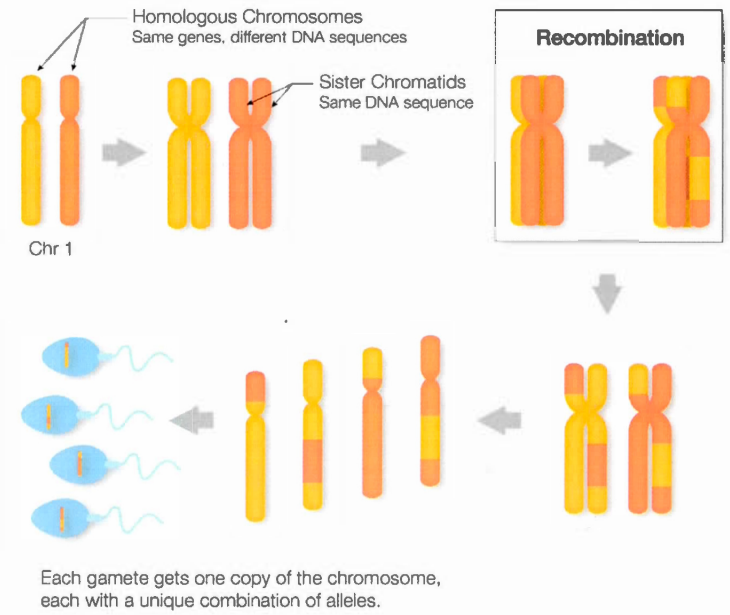


Figure 1.2 Illustration d'une recombinaison. (Center, 2014)

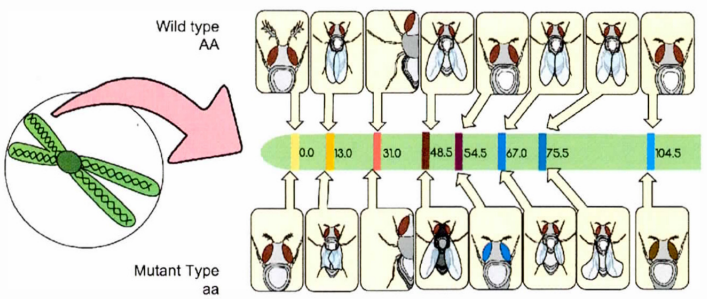


Figure 1.3 Première carte génétique construite par liaisons génétiques du deuxième chromosome d'une mouche drosophile réalisé par Thomas Hunt Morgan. (Mader, 2005, p. 209)

Un des tests utilisés pour analyser une liaison génétique est le score LOD, pour «*logarithm of odds*» en anglais. Ce test statistique développé par Newton Morton permet d'analyser si deux *loci* sont liés. Toutefois, lorsque les maladies possèdent une faible pénétrance, nous allons plutôt utiliser des tests d'association exploitant le déséquilibre de liaison.

1.3.1 Le déséquilibre de liaison

Bien que son nom semble suggérer un lien direct avec le concept des liaisons génétiques, il s'agit d'un concept différent. Le déséquilibre de liaison correspond au fait d'observer une association non aléatoire de la fréquence allélique à deux *loci* différents dans une population. Soit deux *loci* α et β, avec deux haplotypes possibles (a, A et b, B), les probabilités qui en découlent sont données par le tableau 1.3. Le coefficient du déséquilibre de liaison D_{AB} entre ces deux *loci* est alors donné par :

$$D_{AB} = p_{AB} - P_A P_B, \tag{1.4}$$

en posant A et B comme étant deux allèles distincts et lorsque P_B et P_b ne sont pas nulles.

La figure 1.4 illustre bien le rôle des recombinaisons dans le déséquilibre de liaison. Au chapitre suivant, nous décrivons plus profondément le graphe de recombinaison ancestral, qui modélise à l'aide du processus de coalescence une histoire probable d'un ensemble de séquences génétiques. Imaginons maintenant qu'une maladie soit causée par une mutation simple et qu'on échantillonne des séquences provenant d'un site où la population est en état de déséquilibre de liaison. Alors, on pourra tirer un maximum de précision dans l'estimation de la position d'une mutation délétère comparativement à une méthode où aucune inférence sur la phylogénie des séquences n'est effectuée. Les statistiques impliquant l'ensemble des données

Tableau 1.3 Tableau des probabilités pour deux loci. [1]

		Locus β		
		B	b	
Locus α	A	p_{AB}	p_{Ab}	P_A
	a	p_{aB}	p_{ab}	P_a
		P_B	P_b	1

présentes dans un ARG possèdent donc un avantage considérable, car elles exploitent l'information contenue dans la topologie du graphe. Nous détaillerons dans les chapitres subséquents diverses approches possibles pour exploiter l'information contenue dans la topologie des ARGs.

1. Source : `csg.sph.umich.edu/abecasis/class/666.03.pdf`

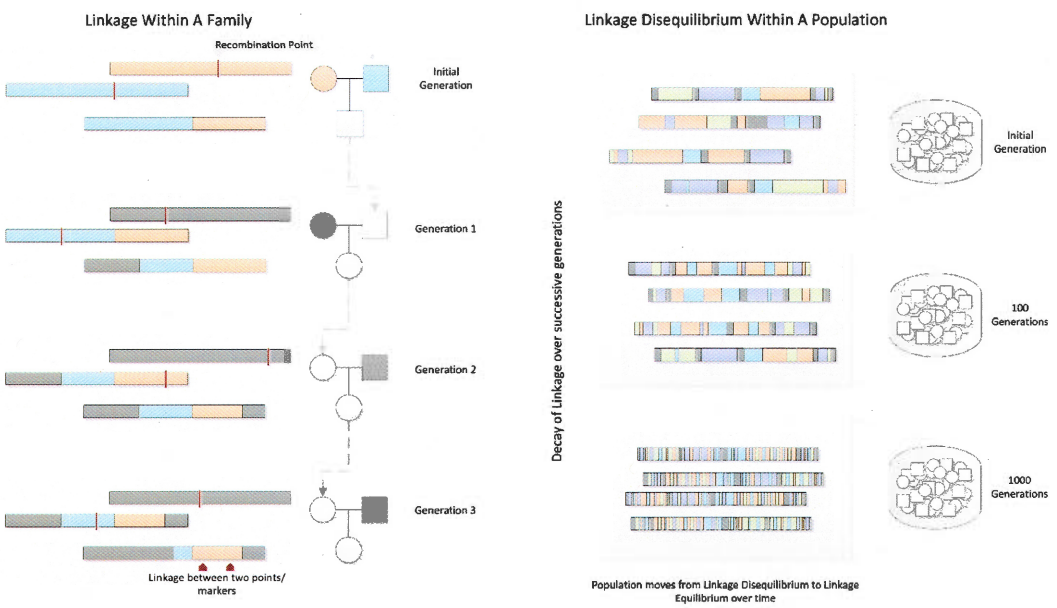


Figure 1.4 Illustration du déséquilibre de liaison et de la recombinaison. On peut apercevoir dans la phylogénie à gauche un endroit dans la séquence où il y a un déséquilibre de liaison. On y illustre aussi le concept du point de vue populationnel dans la partie à droite. (Bush et Moore, 2012, chap. 11)

CHAPITRE II

LES GRAPHES PHYLOGÉNÉTIQUES

Avec la quantité toujours croissante de données génétiques, les techniques phylogéniques se sont raffinées. Il existe une multitude de types et d'algorithmes de construction de graphes phylogénétiques. Certains, comme les graphes de recombinaison ancestraux modélisent très précisément l'évolution de séquences pour un échantillon qui provient d'une seule et même espèce. D'autres techniques s'efforcent d'effectuer une analyse plus macroscopique et produisent des graphes qui peuvent couvrir l'ensemble des domaines du vivant. La figure 2.1 illustre les étapes menant à la construction d'une phylogénie. Dans ce chapitre, nous décrirons sommairement comment les ARGs sont construits, mais auparavant nous présenterons un algorithme plus simple.

2.1 Phylogénie parfaite

Imaginons maintenant que nous voudrions reconstruire l'historique des 4 séquences observées découlant d'une seule souche originelle. Premièrement, nous allons représenter les bases nucléiques en vert du tableau 1.2b par un «1» et celles en rouge par un «0». On dit alors que les «1» représentent l'allèle mutant et les «0» l'allèle primitif. On remarque aussi qu'il y a toujours seulement deux allèles par position et que la souche originelle possède seulement les versions primitives des allèles.

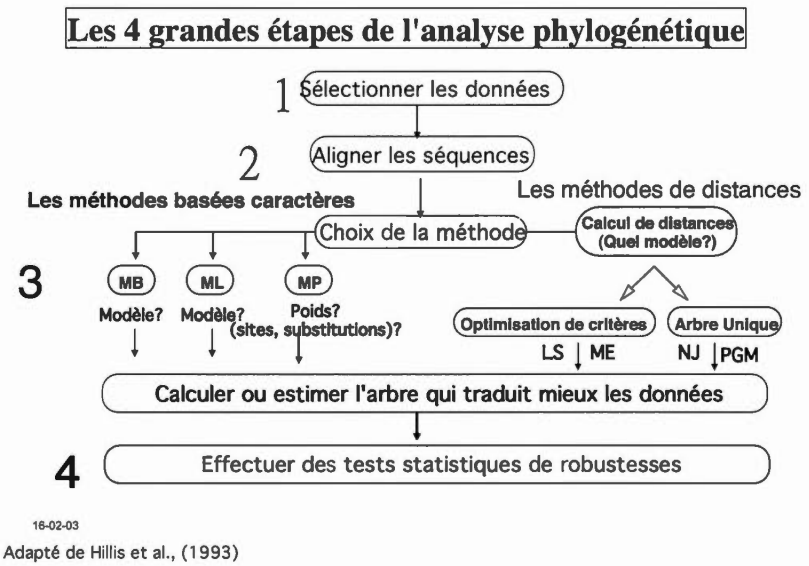


Figure 2.1 Flux normal de la construction d'une phylogénie. [2]

Maintenant, pour inférer une phylogénie, nous pouvons faire les deux hypothèses suivantes :

— il n'y a qu'une seule mutation dans la phylogénie par position ;

— il n'y a aucune mutation indépendante pour chaque allèle mutant.

La première hypothèse suppose qu'il n'y a aucune mutation récurrente. La deuxième hypothèse correspond à l'homologie, c'est-à-dire que les traits observés proviennent d'un même ancêtre commun. Elle est contraire à l'homoplasie où l'évolution peut être convergente, c'est-à-dire que certaines mutations peuvent apparaitre au même *locus* de manière indépendante. Bien que cette dernière hypothèse peut être facilement contestable, lors de la construction de graphe phylogénétique avec des organismes qui proviennent d'une seule et même population cela est raisonnable.

---

2. Source : notes de cours BIF7001 par Matthieu Wilems Ph.D. & Abdoulaye Baniré Diallo Ph.D.

Tableau 2.1 Échantillon de quatre séquences auxquelles une souche originelle qui contient seulement des marqueurs génétiques primitifs a été ajoutée. Le MRCA (pour «most recent common ancestor» en anglais) correspond à la séquence qui deviendra la racine de l'arbre.

Locus	2	6	8	14	17	20
Souche originelle (MRCA) :	0	0	0	0	0	0
Séquence 1 :	1	1	0	0	0	0
Séquence 2 :	0	1	1	1	0	0
Séquence 3 :	0	1	0	1	0	1
Séquence 4 :	0	0	0	0	1	0

Soit M, la matrice contenant les représentations binaires des séquences ainsi que les positions des marqueurs sur la première ligne. Alors, l'algorithme 2.1.1 permet de construire un arbre phylogénique pour les séquences du tableau 2.1. La fonction de tri à la troisième ligne effectue un tri en ordre croissant des colonnes en ne tenant pas en compte la première ligne. Un exemple du résultat de ce tri est présenté dans la figure 2.2. La fonction «$ajouterEspece$» parcourt chaque position de l'entier à partir de la racine de l'arbre, s'il n'y a pas d'arête pour une position, la fonction ajoute un nœud dans l'arbre. Un exemple d'arbre final construit est présenté à la figure 2.2b.

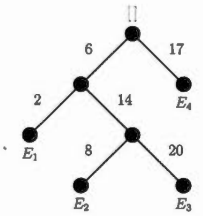
Le résultat des modifications effectuées par l'algorithme 2.1.1 sur les deux différentes structures de données se trouve à la figure 2.2. Pour reconstruire les séquences à partir de l'arbre, il suffit de le parcourir à partir d'une feuille jusqu'à la racine et faire les mutations aux positions indiquées par les arêtes que l'on parcourt. Par exemple, si l'on emprunte l'arête «2» il faut muter le C par un A à la position 2 de la séquence.

Algorithme 2.1.1 Algorithme permettant de construire un arbre phylogénique.

```
1: fonction ConstruireArbrePhylogénique(M(n, m) : Matrice) : un arbre
2:     Arbre arbreResultat ← arbre vide()
3:     Trier M par colonnes ;
4:     Tableau d'entier T ← [ ]
5:     pour i ← 1, 2, . . . , n − 1 faire
6:         T ← [ ]
7:         pour j ← 0, , . . . , m − 1 faire
8:             si M[i, j] = 1 alors
9:                 T ← T + M[0, j]
10:            fin si
11:        fin pour
12:        arbreResultat.ajouterEspece(tableauRepresentantUneEspece)
13:    fin pour
14:    retourner arbreResultat
15: fin fonction
```

Position	6	2	14	8	20	17
MRCA :	0	0	0	0	0	0
Séquence 1 :	1	1	0	0	0	0
Séquence 2 :	1	0	1	1	0	0
Séquence 3 :	1	0	1	0	1	0
Séquence 4 :	0	0	0	0	0	1

(a) La matrice M une fois triée.



(b) L'arbre produit par l'algorithme.

Figure 2.2 Illustration du résultat de l'algorithme 2.1.1.

2.2 Le graphe de recombinaison ancestral

Avec un ensemble de séquences génétiques on infère un graphe complexe modélisant l'évolution des séquences observées depuis un ancêtre commun.

2.2.1 Présentation générale

Pour expliquer naïvement ce que représente un ARG, on pourrait dire que ce graphe est aux chromosomes ce qu'un arbre généalogique est aux individus. Le graphe de la figure 2.3 illustre l'histoire d'un ensemble de séquences chromosomiques observées (les feuilles), jusqu'à son origine (la racine du graphe). Les nœuds internes représentent des séquences intermédiaires qui sont des ancêtres communs d'un sous-ensemble de l'échantillon seulement. Tout ensemble de séquences chromosomiques observées découle d'une seule séquence ayant existé antérieurement. On nomme cette séquence le «MRCA» (pour *«most recent common ancestor»* en anglais). Or, comme il est impossible d'obtenir le code génétique pour tous les individus dans la phylogénie, il faut donc inférer une phylogénie à partir de séquences observées seulement. Notons que pour chacun des *loci* il est possible d'extraire un arbre partiel à partir du graphe.

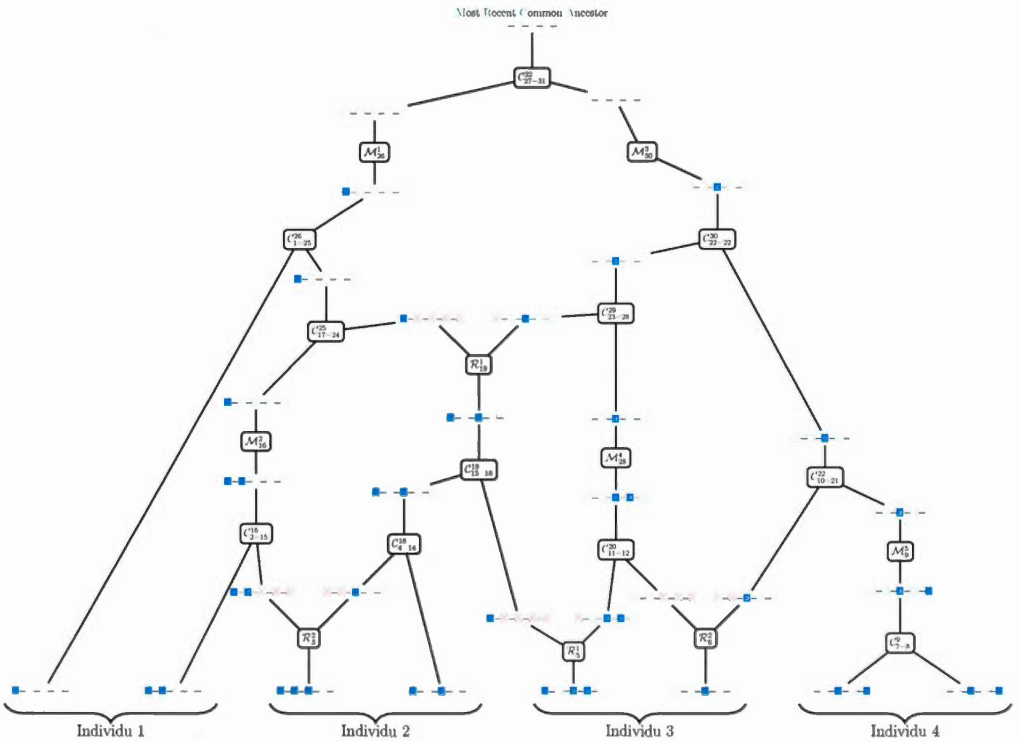


Figure 2.3 Graphe de recombinaison ancestral simple pour 4 individus (diploïdes), donc 8 séquences chromosomiques de 5 marqueurs. Le symbole «□» représente un allèle primitif tandis que le symbole «■» représente un allèle mutant. Le matériel non ancestral introduit lors des recombinaisons est représenté par des «⊠».

2.2.2 Les opérations

Nous présentons brièvement les 3 opérations de bases utilisées pour inférer un graphe de recombinaison ancestral.

2.2.2.1 Les mutations

Les mutations sont traitées d'une manière analogue aux hypothèses utilisées dans l'algorithme de la phylogénie parfaite présenté auparavant. On suppose que pour chaque *locus* il y a eu une seule mutation dans la phylogénie qui a engendré la mutation et qu'il n'y a pas eu d'autre évènement parallèle. On suppose aussi que seulement deux allèles sont possibles par position. Ces suppositions sont très proches du comportement réel des séquences. La figure 2.4 illustre la représentation d'une mutation dans un ARG.

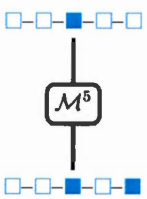


Figure 2.4 Exemple d'une mutation sur le cinquième locus dans un ARG.

2.2.2.2 Les coalescences

Une coalescence permet de fusionner deux nœuds partageant les mêmes allèles connus. Cette opération semblable à un évènement de spéciation découle directement de la théorie de la coalescence que nous aborderons plus loin. Parfois, certaines séquences contiennent du matériel non ancestral qui est introduit par une ou plusieurs recombinaisons. Dans ce cas on ignore ces marqueurs lorsqu'on déterminera si une coalescence est possible entre deux nœuds. La figure 2.5 illustre

une coalescence entre deux séquences contenant du matériel non ancestral.

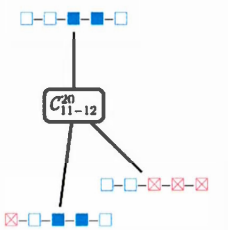


Figure 2.5 Exemple d'une coalescence contenue dans la figure 2.3 entre les séquences 11 et 12 pour générer la séquence 20.

2.2.2.3 Les recombinaisons

Les recombinaisons permettent de transmettre à sa descendance un mélange des deux chromosomes homologues que possède un individu. Lors d'une recombinaison sur un nœud dans un ARG, on crée deux nouveaux nœuds avec les marqueurs situés des deux côtés du point de recombinaison. On introduit alors aux extrémités des séquences produites du matériel non ancestral. Les recombinaisons qui se produisent lors de l'inférence du graphe permettent de faire varier les arbres partiels pour chacun des *loci* qui recevront les mutations d'intérêt selon la position de l'insertion. La figure 2.6 illustre une recombinaison qui introduit du matériel non ancestral.

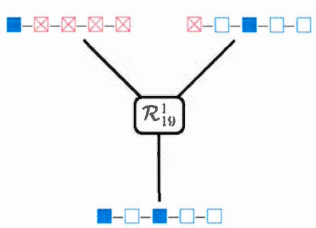


Figure 2.6 Exemple d'une recombinaison contenue dans la figure 2.3 de la séquence avec l'index 19 à la position 1.

2.2.3 Algorithme de construction

La construction d'un ARG quelconque du passé vers le futur est assez aisée à simuler. En effet, il suffit de choisir un des évènements possibles entre une mutation, une coalescence ou une recombinaison en suivant certaines probabilités et de répéter l'opération jusqu'à l'obtention du nombre de séquences désiré. Toutefois, le résultat obtenu ne sera pratiquement jamais identique à un échantillon d'intérêt. C'est pourquoi, en pratique, on construit les ARGs du présent (un échantillon de séquences) vers le passé. Cependant, il s'agit seulement d'une des réalisations possibles et nous ne pourrons jamais confirmer qu'un ARG est bien celui qui représente la réalité. Donc, il est important de définir une métrique qui mesurera la qualité d'un ARG selon un modèle et les hypothèses qu'il comporte. Diverses approches existent, certaines sont plus algorithmiques, d'autres sont plus statistiques.

2.2.3.1 Approche algorithmique

Plusieurs approches algorithmiques d'inférence ont été explorées :

- par séparation et évaluation, ou «*branch and bound*» en anglais (Lyngsø *et al.*, 2005) ;
- par minimisation du nombre de recombinaisons (Song et Hein, 2005) ;
- à l'aide d'heuristique comme Margarita (Minichiello et Durbin, 2006) ou ARG4WG (Nguyen *et al.*, 2017).

Nous avons développé un logiciel *FastARG* [3] qui implémente plusieurs algorithmes de construction, dont Margarita et ARG4WG mentionnés ci-dessus. Nous dé-

3. `https://bitbucket.org/fastarguqam/` - (13 septembre 2018)

taillons dans les chapitres subséquents l'implémentation informatique de ces algorithmes ainsi que certains résultats. Ces méthodes ont l'avantage d'être rapides, mais il n'est pas trivial de quantifier la qualité des graphes obtenus, car ils ne sont pas échantillonnés dans un ensemble de graphes de recombinaison ancestraux possibles.

2.2.3.2 Approche statistique

Les ARGs obtenus à l'aide de méthodes statistiques (Fearnhead et Donnelly, 2001, Griffiths et Marjoram, 1996) peuvent être plus facilement combinés pour être analysés par la suite. Toutefois, ils sont généralement beaucoup plus difficiles (voire impossibles) à générer pour un grand nombre de marqueurs génétiques.

2.3 Utilité des graphes

Outre le fait que ces graphes nous renseignent sur nos origines, ils peuvent aussi avoir des applications très concrètes comme :

- l'estimation du taux de mutation ou de recombinaison ;
- l'identification de la source d'un virus ou une bactérie ;
- la cartographie génétique fine ;
- la position d'une mutation sur un gène ;
- etc.

On entend par cartographie génétique fine le fait de trouver l'endroit précis où une ou plusieurs mutations génétiques influençant le phénotype d'un individu se sont produites. Supposons qu'une maladie dite «simple» serait provoquée par seulement une (ou très peu) de ces mutations. On appellerait alors ces mutations

des «TIM» pour «*trait-influencing mutation*» en anglais. Comme il est possible qu'une seule mutation altère considérablement l'expression d'un gène, il est facile de penser qu'elle pourrait causer une maladie chez le porteur. Pour trouver le *locus* de ces mutations, plusieurs techniques sont possibles. Nous discuterons dans les chapitres qui suivront de quelle manière le graphe de recombinaison ancestral intervient dans le processus de cartographie génétique.

CHAPITRE III

STRATÉGIE COMPUTATIONNELLE

Dans ce chapitre, nous décrivons principalement deux approches possibles pour évaluer les diverses opérations nécessaires pendant la construction d'un ARG. Ces approches ont pour but de minimiser la quantité de calcul requise et donc d'améliorer la complexité temporelle et spatiale des algorithmes qui interviennent dans la création de ces graphes.

3.1 Approche ensembliste

Dans cette approche, nous utilisons les positions des marqueurs et les positions du matériel non ancestral seulement pour effectuer les opérations. Donc une séquence $\mathcal{S}$ est formée de deux ensembles, $\mathcal{M}$, contenant les emplacements des *loci* mutants, et $\mathcal{A}$, contenant les sites du matériel non ancestral :

$$\mathcal{S} = (\mathcal{M}, \mathcal{A}). \tag{3.1}$$

3.1.1 Fonctionnement des coalescences

Lors d'une coalescence, nous devons vérifier qu'aucune position ancestrale des deux séquences (*locus* dont on connait le génotype) ne possède un marqueur incompatible, c'est-à-dire un allèle mutant et un allèle primitif sur le même *locus*.

Dans les deux séquences du tableau 3.1, une coalescence est impossible à cause des *loci* deux et quatre.

Tableau 3.1 Deux séquences de neuf marqueurs qui illustrent toutes les combinaisons possible des paires de loci. Les « ?» représentent les loci non ancestraux.

	1	2	3	4	5	6	7	8	9	$\mathcal{M}$	$\mathcal{A}$
$\mathcal{S}_1$	0	0	0	1	1	1	?	?	?	$\{4,5,6\}$	$\{7,8,9\}$
$\mathcal{S}_2$	0	1	?	0	1	?	0	1	?	$\{2,5,8\}$	$\{3,6,9\}$

On dit que deux séquences $\mathcal{S}_1 = (\mathcal{M}, \mathcal{A})$ et $\mathcal{S}_2 = (\mathcal{M}', \mathcal{A}')$ peuvent coalescer, c'est-à-dire $\mathcal{S}_1 \equiv \mathcal{S}_2$ si et seulement si :

$$|(\mathcal{M} \Delta \mathcal{M}') \setminus (\mathcal{A} \cup \mathcal{A}')| = 0, \tag{3.2}$$

où Δ est la différence symétrique entre deux ensembles. Notons que parfois, selon la définition utilisée, on doit aussi vérifier qu'il existe au moins un *locus* ancestral entre les deux séquences. En définissant $\mathcal{A}^C$ comme étant l'ensemble des *loci* ancestraux d'une séquence, il suffit de vérifier que l'intersection des deux ensembles complémentaires n'est pas vide, c'est-à-dire $\mathcal{A}^C \cap \mathcal{A}'^C \neq \emptyset$. Il est aussi possible d'itérer sur des ensembles ordonnés et vérifier qu'une position est commune.

Comme tous les neuf couples de *loci* possibles sont présents pour les deux séquences $\mathcal{S}_1$ et $\mathcal{S}_2$ dans le tableau 3.1, il est facile de vérifier que l'équation (3.2) est satisfaite. En effet, comme seules les positions deux et quatre sont ancestrales et ont des allèles différents, on devrait les voir apparaitre dans les calculs de l'équi-

valence :

$$\begin{aligned}|(\mathcal{M}_1 \Delta \mathcal{M}_2) \setminus (\mathcal{A}_1 \cup \mathcal{A}_2))| &= |(\{4,5,6\}\Delta\{2,5,8\}) \setminus (\{7,8,9\} \cup \{3,6,9\})| \\ &= |\{2,4,6,8\} \setminus \{3,6,7,8,9\}| \\ &= |\{2,4\}| \\ &= 2 \neq 0 \qquad \Rightarrow \mathcal{S}_1 \not\equiv \mathcal{S}_2.\end{aligned}$$

Cette approche rend le calcul du résultat d'une coalescence trivial. Si on suppose que deux séquences sont compatibles, c'est-à-dire que si $\mathcal{S}_1 \equiv \mathcal{S}_2$, alors la nouvelle séquence $\mathcal{S}_{1+2}$ peut être calculée de la manière suivante :

$$\begin{aligned}\mathcal{S}_{1+2} &= \mathcal{S}_1 + \mathcal{S}_2 \\ &= (\mathcal{M}_1, \mathcal{A}_1) + (\mathcal{M}_2, \mathcal{A}_2) \\ &= (\mathcal{M}_1, \mathcal{A}_1 \cap \mathcal{A}_2) \\ &= (\mathcal{M}_2, \mathcal{A}_1 \cap \mathcal{A}_2) \\ &= (\mathcal{M}_{1+2}, \mathcal{A}_{1+2}).\end{aligned}$$

Comme le matériel non ancestral qui subsistera après une coalescence sera composé uniquement des *loci* non ancestraux communs, alors clairement $\mathcal{A}_1 \cap \mathcal{A}_2 = \mathcal{A}_{1+2}$. Et puisque $\mathcal{S}_1 \equiv \mathcal{S}_2$, alors $\mathcal{M}_{1+2} = \mathcal{M}_1 = \mathcal{M}_2$.

3.1.2 Fonctionnement des mutations et des recombinaisons

Avec cette approche les mutations et les recombinaisons sont faciles à générer. Dans le cas d'une mutation, il suffit d'enlever ou d'ajouter la position du marqueur dans l'ensemble $\mathcal{M}$. Pour une recombinaison, on peut simplement greffer les *loci* non ancestraux selon le point de recombinaison dans l'ensemble $\mathcal{A}$. On peut optionnellement éliminer les marqueurs mutants qui sont devenus des *loci* non ancestraux.

3.1.3 Avantages et désavantages de cette approche

Cette approche est avantageuse si le ratio du nombre de marqueurs mutants sur le nombre de marqueurs total est très faible, minimalement sous la barre des 10%. De cette manière, on gagne en complexité spatiale et temporelle de calcul, car la représentation compressera grandement les séquences. Les opérations génétiques à effectuer sont aussi faciles à traduire en langage ensembliste. Le format comporte aussi beaucoup d'autres avantages, comme le fait de connaître le nombre de SNPs mutants pour une séquence en temps constant. Mais les structures de données et les algorithmes effectuant les opérations ensemblistes sont lourds pour la tâche. En particulier, les 3 opérations ensemblistes utilisées dans l'équation 3.2 ont une complexité temporelle linéaire selon la taille des ensembles. Donc le coût des opérations liées aux coalescences devient vite prohibitif.

3.2 Approche par fonction booléenne

Une fonction booléenne est de la forme $f : \mathbf{B}^k \rightarrow \mathbf{B}$, où $\mathbf{B} = \{0, 1\}$ et $k \geq 0$. Toute fonction booléenne peut être traduite en formule propositionnelle. Nous utiliserons l'algèbre de Boole pour exprimer nos fonctions logiques. La figure 3.2 illustre les différentes opérations possibles dans ce paradigme et les équivalents dans le langage *C/C++*.

Tableau 3.2 Équivalence entre les symboles logiques et les opérations bit à bit correspondantes en C/C++.

		Logique	*C, C++*
Négation	(not)	$\neg$	$\sim$
Conjonction	(and)	$\wedge$	&
Disjonction	(or)	$\vee$	|
Ou exclusif	(xor)	$\oplus$	^

Les fonctions de bases de la logique booléenne (la conjonction, la disjonction, la négation, etc.) sont implémentées de façon matérielle dans les processeurs modernes. La conversion des opérations génétiques sous cette forme permet de minimiser considérablement le temps de calcul. Nous avons donc encodé les séquences avec la plus basse complexité spatiale possible et formuler des fonctions très efficaces. L'avenue de l'algèbre booléenne combinée aux opérations binaires disponibles sur les types de données primitifs de plusieurs langages informatiques constituait une piste naturelle que nous avons explorée.

3.2.1 Encodage des séquences

Comme chaque *locus* des séquences peut prendre trois formes possibles (primitif, mutant ou non ancestral), nous avons besoin de deux bits d'encodage pour chaque position. Donc, il est toujours envisageable d'ajouter un état, car deux bits encodent quatre états possibles, soit 00, 01, 10 et 11 tel qu'illustré dans le tableau 3.3. Le premier bit est un masque qui indique si la position est connue ou non. Puis l'autre bit détermine l'état d'un allèle.

Tableau 3.3 Équivalence des états possibles. Un allèle primitif sera représenté par la paire $(0,0)$*, un allèle mutant par le couple* $(1,0)$ *et les loci non ancestraux sont donnés par les couples* $(0,1)$ *et* $(1,1)$.

$(\mathcal{M}[i], \mathcal{A}[i])$	caractère	symbolique
(0,0)	0	□
(1,0)	1	■
(0,1)	?	⊠
(1,1)	?	⊠

3.2.2 Fonctionnement des coalescences

3.2.2.1 Test de coalescence

Comme dans l'approche précédente nous devons nous assurer que les marqueurs ancestraux non compatibles feront échouer le test de coalescence. Le tableau 3.4 reprend les 16 possibilités de la table 3.1, mais assigne la valeur vraie aux quadruplets qui doivent faire échouer le test.

Après avoir établi une table de vérité, nous devons trouver une fonction booléenne qui donne le résultat que l'on souhaite. Comme il existe 2^{2^k} fonctions binaires pour tout k, une méthode de recherche pour la fonction s'impose. En utilisant les tables de Karnaugh, on peut obtenir facilement une fonction qui nous donne le bon résultat. La table de Karnaugh représenté par le tableau 3.5 est équivalente à l'opération $\mathcal{S} + \mathcal{S}'$ du tableau 3.4.

Tableau 3.4 Table de vérité qui vérifie qu'aucun locus ancestral mutant ne coalesce avec un locus mutant. Dans les deux cas où cela se produit, on donne la valeur vraie. La dernière colonne sert seulement dans le cas où la valeur n'a pas d'impact sur la fonction booléenne que l'on cherche à partir de cette table.

	$\mathcal{S}$		$\mathcal{S}'$		$\mathcal{Y}$		
	$\mathcal{M}$	$\mathcal{A}$	$\mathcal{M}'$	$\mathcal{A}'$	F	V	x
1	0	0	0	0	√		
2	0	0	0	1	√		
3	0	0	1	0		√	
4	0	0	1	1	√		
5	0	1	0	0	√		
6	0	1	0	1	√		
7	0	1	1	0	√		
8	0	1	1	1	√		
9	1	0	0	0		√	
10	1	0	0	1	√		
11	1	0	1	0	√		
12	1	0	1	1	√		
13	1	1	0	0	√		
14	1	1	0	1	√		
15	1	1	1	0	√		
16	1	1	1	1	√		

Tableau 3.5 Table de Karnaugh permettant d'établir une nouvelle formulation du test d'équivalence entre deux séquences pour une coalescence.

$\mathcal{M}\mathcal{A}$ \ $\mathcal{M}'\mathcal{A}'$	00	01	11	10
00	0	0	0	1
01	0	0	0	0
11	0	0	0	0
10	1	0	0	0

Posons $\mathcal{S}[i]$ le ième couple $(\mathcal{M}[i], \mathcal{A}[i])$ de la séquence k la longueur d'une séquence telle que $k = |\mathcal{S}|$. Soit $f : \mathcal{B}^4 \rightarrow \mathcal{B}$, où $\mathcal{B}^4$ correspond à un *locus* pour couple de deux séquences c'est-à-dire : $\mathcal{B}^4 \sim \mathcal{S}[i] \times \mathcal{S}'[i] \sim \mathcal{M}[i] \times \mathcal{A}[i] \times \mathcal{M}'[i] \times \mathcal{A}'[i]$. La fonction booléenne f qui permet de solutionner la table de Karnaugh représenté par le tableau 3.5 est donnée par la formule suivante :

$$(\neg\mathcal{A}[i]) \wedge (\neg\mathcal{A}'[i]) \wedge (((\neg\mathcal{M}[i]) \wedge \mathcal{M}'[i]) \vee (\mathcal{M}[i] \wedge (\neg\mathcal{M}'[i]))) . \tag{3.3}$$

On peut vérifier facilement que l'équation 3.3 donne bien «1» pour le *locus* 3 :

$$\begin{aligned} f(\mathcal{S}[3], \mathcal{S}'[3]) &= (\neg\mathcal{A}[3]) \wedge (\neg\mathcal{A}'[3]) \wedge (((\neg\mathcal{M}[3]) \wedge \mathcal{M}'[3]) \vee (\mathcal{M}[3] \wedge (\neg\mathcal{M}'[3]))) \\ &= (\neg 0) \wedge (\neg 0) \wedge (((\neg 0) \wedge 1) \vee (0 \wedge (\neg 1))) \\ &= 1 \wedge 1 \wedge ((1 \wedge 1) \vee (0 \wedge (0))) \\ &= 1 \wedge 1 \wedge (1 \vee 0)) \\ &= 1 \wedge 1 \wedge 1 = 1 \end{aligned}$$

On retrouvera en annexe des démonstrations supplémentaires pour les *loci* 1, 9 et 16.

En appliquant f sur tous les *loci*, on obtient un vecteur binaire dans $\mathcal{B}^k$ si $|S| = k$. Pour savoir si une coalescence est possible, plusieurs stratégies sont envisageables. Soit on itère sur le vecteur en vérifiant si la valeur vraie (ou 1) est présente pour rejeter la coalescence, soit on décompose le vecteur en bloc et on regarde si la valeur en base 10 le développement binaire est différent de 0. En effet, un seul «1» dans l'expansion binaire suffit pour que le bloc soit différent de zéro. Rappelons que notre table de vérité est conçue pour qu'un «1» apparaisse si un *locus* est incompatible.

Formellement, on définit la fonction f', qui vérifie si une coalescence est possible entre deux séquences de la manière suivante :

$$f' : \quad \mathcal{B}^4 \longrightarrow \mathcal{B}$$

$$f'(\mathcal{S}^2) \mapsto \begin{cases} 1 & \text{si } f(\mathcal{S}[i] \times \mathcal{S}'[i]) = 0 \qquad \forall i \in [0, |\mathcal{S}|], \\ 0 & \text{sinon.} \end{cases}$$

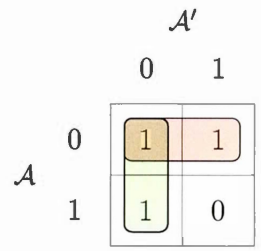
En somme, on a que :

$$\mathcal{S} \equiv \mathcal{S}' \iff f'(\mathcal{S}[i], \mathcal{S}'[i]) = 1.$$

Comme dans l'approche ensembliste précédente, la définition d'une coalescence peut parfois exiger qu'au moins un *locus* commun soit ancestral aux deux séquences. Dans ce cas, nous avons besoin de la table de vérité suivante :

	$\mathcal{S}$	$\mathcal{S}'$		$\mathcal{Y}$	
	$\mathcal{A}$	$\mathcal{A}'$	F	V	x
1	0	0		✓	
2	0	1		✓	
3	1	0		✓	
4	1	1	✓		

(a) Table de vérité qui vérifie si au moins un locus commun est ancestral.



(b) Équivalent sous la forme d'une table de Karnaugh.

Figure 3.1 Tableaux permettant de trouver une fonction booléenne qui indique si au moins un des loci entre deux séquences est ancestral.

La fonction booléenne g qui permet de solutionner la table de Karnaugh à la figure 3.1b est donnée par :

$$g(i) = (\neg\mathcal{A}[i]) \oplus (\neg\mathcal{A}'[i]). \tag{3.4}$$

Formellement, on définit une fonction g', qui vérifie si un *locus* est commun à deux séquences de la manière suivante :

$$g' : \quad \mathcal{B}^2 \longrightarrow \mathcal{B}$$

$$g'(\mathcal{A}^2) \mapsto \begin{cases} 1 & \text{si } \exists i \in [0, |\mathcal{A}|] \mid g(\mathcal{A}[i], \mathcal{A}'[i]) \neq 0, \\ 0 & \text{sinon.} \end{cases}$$

Enfin, on peut définir une coalescence stricte de la manière suivante :

$$\mathcal{S} \equiv \mathcal{S}' \iff f'(\mathcal{S}[i], \mathcal{S}'[i]) \wedge g'(\mathcal{A}[i], \mathcal{A}'[i]) = 1. \tag{3.5}$$

3.2.2.2 Calcul d'une coalescence

Puisque nous sommes en mesure de savoir si deux séquences peuvent coalescer, nous devons trouver une fonction qui génère une nouvelle séquence à partir de deux séquences sources. Soit la fonction $+ : \mathcal{S} \times \mathcal{S} \longrightarrow \mathcal{S}$ qui coalesce deux séquences ensemble. Idéalement, on pourrait définir une addition de la manière suivante :

$$\begin{aligned} +: \quad & \mathcal{S}^2 \longrightarrow \mathcal{S} \\ & \mathcal{S}_1[i] + \mathcal{S}_2[i] \longmapsto (\mathcal{M}_1[i], \mathcal{A}_1[i] \wedge \mathcal{A}_2[i]) = (\mathcal{M}_2[i], \mathcal{A}_1[i] \wedge \mathcal{A}_2[i]). \end{aligned}$$

Malheureusement, le test de coalescence de l'équation 3.5 ne permet pas d'utiliser cette fonction. En effet, les lignes 10 et 13 du tableau 3.4 nous donnent le résultat suivant :

$$\begin{aligned} (\mathcal{M}_1[10], \mathcal{A}_1[10] \wedge \mathcal{A}_2[10]) &= (1, 0 \wedge 1) \\ &= (0, 0) \\ &\neq (1, 0) \\ &= (\mathcal{M}_2[10], \mathcal{A}_1[10] \wedge \mathcal{A}_2[10]). \end{aligned}$$

On obtient un résultat similaire pour la ligne 13. Puisque nous n'avons aucune garantie qu'un allèle mutant n'est pas caché par une position non ancestrale nous devons nous assurer que l'allèle provient toujours de la séquence ancestrale pour tous les *loci*.

Tableau 3.6 La table de vérité pour le calcul d'une coalescence stricte. Le calcul de l'ensemble $\mathcal{A}_{1+2}$ correspond à un «et» logique sur les séquences $\mathcal{A}$ d'origine. On peut voir aussi la correspondance symbolique utilisée pour représenter l'encodage binaire.

	$\mathcal{S}_1$		$\mathcal{S}_2$		$\mathcal{M}_{1+2}$	$\mathcal{A}_{1+2}$	Y			$\mathcal{C}_{1-2}^{1+2}$
$(\mathcal{B}^4)_{10}$	$\mathcal{M}_1$	$\mathcal{A}_1$	$\mathcal{M}_2$	$\mathcal{A}_2$	$(\mathcal{M}_1 \wedge \neg\mathcal{A}_1) \vee (\mathcal{M}_2 \wedge \neg\mathcal{A}_2)$	$\mathcal{A}_1[i] \wedge \mathcal{A}_2[i]$	F	V	x	$\mathcal{S}_1 \oplus \mathcal{S}_2 = \mathcal{S}_{1+2}$
0	0	0	0	0	0	0	√			□ +□ = □
1	0	0	0	1	0	0		√		□ +⊠ = □
2	0	0	1	0	1	0			√	□ +■ ≠ ■
3	0	0	1	1	0	0		√		□ +⊠ = □
4	0	1	0	0	0	0	√			⊠ +□ = □
5	0	1	0	1	0	1	√			⊠ +⊠ = ⊠
6	0	1	1	0	1	0		√		⊠ +■ = ■
7	0	1	1	1	0	1	√			⊠ +⊠ = ⊠
8	1	0	0	0	1	0			√	■ +□ ≠ ■
9	1	0	0	1	1	0		√		■ +⊠ = ■
10	1	0	1	0	1	0		√		■ +■ = ■
11	1	0	1	1	1	0		√		■ +⊠ = ■
12	1	1	0	0	0	0	√			⊠ +□ = □
13	1	1	0	1	0	1	√			⊠ +⊠ = ⊠
14	1	1	1	0	1	0		√		⊠ +■ = ■
15	1	1	1	1	0	1	√			⊠ +⊠ = ⊠

Les deux valeurs «x» sur les lignes 2 et 8 du tableau 3.6 correspondent au fait que l'on peut ignorer le résultat produit dans le calcul de la fonction booléenne à partir de la table de vérité. En effet, comme une coalescence est possible si et seulement si $\mathcal{S}_1 \equiv \mathcal{S}_2$, on sait que ces cas de figure ne se présenteront pas. Cela permet de minimiser la fonction booléenne et par le fait même, diminuer la quantité de calcul nécessaire pour une coalescence.

Donc, on définit l'opérateur de la manière suivante :

$$\begin{aligned} +: \quad & \mathcal{S}^2 \longrightarrow \mathcal{S} \\ & \mathcal{S}_1[i] + \mathcal{S}_2[i] \longmapsto ((\mathcal{M}_1 \wedge \neg\mathcal{A}_1) \vee (\mathcal{M}_2 \wedge \neg\mathcal{A}_2), \mathcal{A}_1[i] \wedge \mathcal{A}_2[i]). \end{aligned} \tag{3.6}$$

De cette manière, si $S_1 \equiv S_2$ le résultat sera cohérent dans tout les cas.

3.2.3 Fonctionnement des mutations et des recombinaisons

Dans le cas des mutations, il s'agit de changer l'état du bit à la position désirée dans l'ensemble $\mathcal{M}$. Les recombinaisons introduisent du matériel non ancestral et conservent le statut des *loci* mutants. On peut donc simplement casser la séquence à l'endroit désiré, puis ajouter des marqueurs neutres sur les bouts des deux séquences produites et marquer ces *loci* comme non ancestraux. Ces opérations sont relativement simples à implémenter.

3.2.4 Avantages et désavantages de cette approche

Un des seuls avantages est la rapidité de traitement des opérations par un ordinateur. Toutefois, cette représentation, bien qu'efficiente en termes d'espace et de temps de calcul, complexifie considérablement toutes les opérations sur les séquences. En effet, lors du calcul d'un ARG, plusieurs autres opérations doivent être implémentées, et le processus de recherche afin de trouver la bonne stratégie pour convertir un algorithme sémantique de haut niveau en fonction booléenne n'est pas trivial.

CHAPITRE IV

DÉRIVE GÉNÉTIQUE

L'idée que des phénomènes aléatoires soient responsables en partie de l'évolution et de la diversité des espèces est parfois surprenante. Plusieurs modèles de dérive génétique (*«Genetic drift»* en anglais) tentent de reproduire le plus fidèlement possible la variation que l'on observe dans une population ou un ensemble de séquences dans le but d'expliquer et de mieux comprendre l'évolution. Certains modèles comme la théorie de la coalescence sont rétrospectifs, c'est-à-dire que l'on tente de voir l'évolution en remontant dans le temps. D'autres, comme le modèle de Wright–Fisher, sont prospectifs, car ils déduisent l'historique de la variation allélique d'une population d'origine vers le futur. Dans ce chapitre, nous décrivons sommairement ces deux modèles.

4.1 Le modèle Wright–Fisher

Un des modèles les plus simples pour illustrer la variation de la fréquence allélique d'une population a été imaginé d'abord par Ronald Fisher (Fisher, 1930) et formalisé par Sewall Wright (Wright, 1931) dans les années trente. Bien que le modèle fait fi de plusieurs détails importants, on peut s'en servir pour générer des phylogénies facilement. De plus, il forme la base théorique sur laquelle nous introduirons des modèles plus complexes. Ce modèle a pour but de décrire la variation

allélique dans une population idéalisée lors de son évolution. Voici une liste non exhaustive des hypothèses faites par le modèle :

— un seul *locus* ayant seulement deux allèles possibles par individu ;

— aucune mutation, recombinaison ou sélection ;

— la taille de la population est constante ;

— aucun chevauchement des générations.

Posons H_0, la première génération comportant N individus diploïdes ou $2N$ individus haploïdes ayant chacun un des deux allèles possibles, notés α ou β. Pour créer la génération H_1 suivante, on choisit tous les $2N$ individus aléatoirement parmi ceux présents à la génération H_0. On répète le processus pour les générations subséquentes. La probabilité d'obtenir k allèles α à la génération H_{t+1} si $P(\alpha) = p$ à la génération H_t est donnée par l'équation suivante :

$$\binom{2N}{k} p^k (1-p)^{2N-k} \sim \mathcal{B}(2N, p). \tag{4.1}$$

On peut alors se demander combien de générations sont nécessaires avant qu'un des allèles soit complètement fixé. C'est-à-dire qu'on cherche le nombre de générations τ pour que $P(\alpha) = 0$ ou $P(\alpha) = 1$. Notons que si $P(\alpha) = 0$, alors $P(\beta) = 1$ et vice versa car on suppose que nous avons seulement deux allèles possible. Posons $\{X_t\}_{t \geq 0}$ le nombre d'allèles α à la génération t, alors le temps de fixation est donné par le théorème suivant :

Théorème 4.1. [1] *La probabilité qu'un allèle soit fixé au temps $t = \tau$ sachant que la fréquence allèlique initiale est donnée par :*

$$\mathbb{P}[X_\tau = 2N \mid X_0 = i] = \frac{i}{2N} \tag{4.2}$$

1. `https://www.math.wisc.edu/~roch/evol-gen/roch-evolgen-notes15.pdf`

Démonstration. On a que la fréquence allèlique pour la génération suivante en fonction de la génération courante est :

$$\mathbb{P}[X_{t+1} = k|X_t] = \binom{2N}{k} \frac{X_t}{2N}^k \left(1 - \frac{X_t}{2N}\right)^{2N-k} \text{ où } X_{t+1}|X_t \sim \mathcal{B}\left(2N, \frac{X_t}{2N}\right).$$

Remarquons que si $X_t = 0$ ou $2N$, alors la probabilité de l'équation ci-haut est nulle car les états sont absorbants et la fréquence allèlique est fixée. Le temps de fixation τ est défini de la manière suivante :

$$\tau = \min\{t : X_t = 0 \text{ ou } 2N\}.$$

Comme $X_{t+1}|X_t \sim \mathcal{B}\left(2N, \frac{X_t}{2N}\right)$, alors :

$$\mathbb{E}[X_{t+1} \mid X_0, ..., X_t] = \mathbb{E}[X_{t+1} \mid X_t] = 2N\frac{X_t}{2N} = X_t \quad \forall t, \tag{4.3}$$

et puisque $\mathbb{E}(|X_t|) < \infty$ pour tout t, alors $\{X_t\}_{t\geq 0}$ est une martingale, on a par induction :

$$\begin{aligned}
\mathbb{E}[X_t] &= \mathbb{E}[\mathbb{E}[X_t|X_{t-1}]] && \text{(Espérance totale)} \\
&= \mathbb{E}[X_{t-1}] && \text{(par 4.3)} \\
&= \mathbb{E}[X_{t-2}] \\
&= ... \\
&= \mathbb{E}[X_1].
\end{aligned}$$

À l'aide de l'égalité ci-haut, on a que :

$$i = \mathbb{E}[X_1|X_0 = i] = \frac{\mathbb{E}[X_1]}{\mathbb{P}[X_0 = i]} = \frac{\mathbb{E}[X_t]}{\mathbb{P}[X_0 = i]} = \mathbb{E}[X_t|X_0 = i]. \tag{4.4}$$

Finalement, comme on sait que τ converge, alors en posant $t = \tau$ on obtient notre résultat car :

$$\begin{aligned}
\mathbb{P}[X_\tau = 2N \mid X_0 = i] &= \frac{2N\mathbb{P}[X_\tau = 2N \mid X_0 = i]}{2N} \\
&= \frac{\mathbb{E}[X_\tau \mid X_0 = i]}{2N} \\
&= \frac{i}{2N}.
\end{aligned}$$

□

Ce modèle permet aussi de générer des simulations d'évolution pour une population. Il suffit de voir le modèle de Wright-Fisher comme la sélection aléatoire d'individus sélectionnés avec remise dans la population de la génération antérieure. Comme il s'agit d'un modèle prospectif, on crée d'abord des générations d'individus du présent vers le futur. On note ces générations $H_0, H_1, ..., H_n$. Pour passer d'une génération à la suivante, on assigne comme parent à chacun des $2N$ individus de H_{t+1} un des individus de H_t. Ceux qui n'auront pas été choisis dans H_t ne se sont pas reproduits en quelque sorte. Puis en regardant les liens rétrospectivement, on peut facilement extraire une phylogénie.

Supposons maintenant que l'on souhaite analyser l'évolution d'un sous-ensemble de la population à la génération H_0. Puisque la probabilité de choisir un individu particulier à la génération précédente est $\frac{1}{2N}$, alors la probabilité que deux individus possèdent le même parent est $\frac{1}{(2N)^2}$. Comme il existe toujours un lien entre les individus d'une génération quelconque vers la génération précédente, on sait que la probabilité que deux individus n'aient pas trouvé de parent commun après t générations est : $\left(1 - \frac{1}{2N}\right)^t$. Donc la probabilité qu'au moins deux séquences aient trouvé un ancêtre commun à la génération t est : $1 - \left(1 - \frac{1}{2N}\right)^t$. La probabilité de trouver un parent commun exactement à la $k^{ième}$ génération équivaut à dire qu'on n'a pas trouvé de parent commun dans les $k-1$ première génération et que la dernière a réussie. Soit i et j, deux individus quelconques, la probabilité de trouver un parent commun à la génération k est donnée par l'équation suivante :

$$\mathbb{P}[\mathcal{C}_{ij}^k] = \left(1 - \frac{1}{2N}\right)^{k-1} \left(\frac{1}{2N}\right). \tag{4.5}$$

Notons que la probabilité que deux séquences ne trouvent pas d'ancêtres communs au temps $t = 2N$ et lorsque la population tend vers l'infini est :

$$\lim_{N\to\infty} \left(1 - \frac{1}{2N}\right)^{2N} = e^{-1} \approx 0.367879. \tag{4.6}$$

Autrement dit, si la population est infinie, selon ce modèle, le temps qu'une séquence trouve un ancêtre commun est donné par la distribution exponentielle avec $\lambda = 1$.

4.2 Théorie de la coalescence

La théorie de la coalescence est similaire au modèle de Wright-Fisher, mais à la différence qu'elle modélise l'historique du présent vers le passé. Une coalescence correspond au fait de trouver un ancêtre commun à deux individus. Dans les graphes de niveau interspécifique, on peut interpréter une coalescence comme un évènement de spéciation. Au niveau interpopulationnel, on dit que la séquence résultant de la coalescence de deux individus est le plus récent des ancêtres communs pour ces deux séquences. On peut alors facilement généraliser cette notion pour un ensemble de séquences génétiques. De manière analogue au modèle de Wright-Fisher, la théorie de la coalescence implique que la probabilité de voir une coalescence à la génération t suit une distribution géométrique avec $p = 1/2N$. Si N est suffisamment grand, on peut approximer avec la loi exponentielle de la manière suivante :

$$\mathbb{P}[\mathcal{C}_{ij}^{t-1}] = \left(1 - \frac{1}{2N}\right)^{t-1}\left(\frac{1}{2N}\right) \approx \frac{1}{2N} e^{-\frac{t-1}{2N}} \sim \exp\left(\frac{1}{2N}\right), \qquad (4.7)$$

où $\mathcal{C}_{ij}$ correspond à un évènement de coalescence entre deux individus i et j appartenant à la génération $t-1$. Puisque la distribution exponentielle est continue, le passage vers cette distribution à l'équation (4.7) permet d'obtenir des temps pour les coalescences sur une échelle continue. Lorsqu'on génère des phylogénies, nous devons simuler des intervalles de temps avant que le prochain évènement de coalescence ne survienne. Pour ce faire, on doit connaitre la distribution des temps de coalescence. On a que la probabilité que deux individus possèdent un parent

distinct est :

$$\mathbb{P}[\text{2 parents distincts}] = 1 - \left(2N\frac{1}{(2N)^2}\right) = 1 - \frac{1}{2N}. \tag{4.8}$$

Mais quelle est la probabilité $\mathbb{P}[\text{3 parents distincts}]$? Clairement on a que :

$$\mathbb{P}[\text{3 parents distincts}] = \mathbb{P}[\text{2 parents distincts}] \cdot \mathbb{P}[3^{\text{ième}} \text{ distincts des 2 autres}]. \tag{4.9}$$

Pour choisir le troisième parent, nous devons le choisir parmi les $\frac{2N-2}{2N} = 1 - \frac{2}{2N}$ restants. Donc, la probabilité d'obtenir n parents distincts à la génération précédente est donnée par l'équation suivante :

$$\mathbb{P}[n \text{ parents distincts}] = \prod_{i=1}^{n-1}\left(1 - \frac{i}{N}\right) \approx 1 - \frac{n(n-1)}{4N}. \tag{4.10}$$

Donc, on peut généraliser l'équation (4.7) pour avoir des générations comportant des parents distincts seulement. On a alors que la probabilité qu'il y ait une coalescence à la $k^{ème}$ génération est :

$$\mathbb{P}[\mathcal{C}_{i,j}^k|n] = (\mathbb{P}[n \text{ parents distincts}])^{k-1}(1-\mathbb{P}[n \text{ parents distincts}]) \approx \exp\left(\frac{n(n-1)}{4N}\right). \tag{4.11}$$

On peut ainsi faire des tirages aléatoires dans cette distribution exponentielle pour générer des temps de coalescence aléatoire. Le tableau 4.1 résume les équations importantes pour générer une phylogénie basée sur la théorie de la coalescence.

Maintenant que le modèle est bien établi, il serait intéressant de regarder combien de générations sont nécessaires pour arriver au MRCA en fonction de la taille de la population et du nombre de lignées que l'on souhaite voir converger. Le temps

Tableau 4.1 Résumé des équations reliées à la simulation d'une phylogénie selon la théorie de la coalescence. Notons que la fonction $T(i)$ correspond à un tirage aléatoire dans la distribution $C[t|n=i]$.

Description	Équation
Probabilité qu'aucune coalescence ne soit survenue pendant k générations.	$\left(1-\frac{1}{2N}\right)^k$
Probabilité que tous les individus aient des ancêtres distincts à la génération précédente.	$\mathbb{P}[\text{n parents distincts}] \approx 1-\frac{n(n-1)}{4N}$
Distribution des temps d'attente entre les coalescences.	$C[t\|n]=\frac{n(n-1)}{4N}e^{-\left(\frac{n(n-1)}{4N}\right)t}$
Temps moyen d'attente avant une coalescence.	$\mathbb{E}\left[T(C[t\|n])\right]=\frac{1}{\lambda}=\frac{4N}{i(i-1)}$
Temps pour arriver au MRCA.	$T(MRCA)=\sum_{i=2}^{n}T[i]$

moyen pour arriver au MRCA sera donné par l'équation suivante :

$$\begin{aligned}\mathbb{E}[T[MRCA]] &= \mathbb{E}\left[\sum_{i=2}^{n}T[i]\right] \\ &= \sum_{i=2}^{n}\mathbb{E}[T[i]] \\ &= \sum_{i=2}^{n}\frac{4N}{i(i-1)} \\ &= 4N\sum_{i=2}^{n}\frac{1}{i(i-1)} \\ &= \frac{4N(n-1)}{n}.\end{aligned} \tag{4.12}$$

En posant que l'unité de temps t pour chaque génération est $2N$, on obtient le tableau 4.2 qui illustre les temps moyens avant d'arriver au MRCA selon différents scénarios.

Tableau 4.2 Temps moyen avant d'obtenir le MRCA selon plusieurs scénarios différents. Notons que le temps moyen augmente proportionnellement selon la taille de la population. Toutefois, le temps avant d'arriver au MRCA est moins sensible aux nombres d'individus dans la population.

	$\mathbb{E}[T[MRCA]]$		
n	$N = 100$	$N = 500$	$N = 1000$
2	200,0	1000,0	2000,0
3	266,7	1333,3	2666,7
5	320,0	1600,0	3200,0
10	360,0	1800,0	3600,0
20	380,0	1900,0	3800,0

Nous avons tous les éléments en place pour concevoir un algorithme qui permet de générer une phylogénie selon la théorie de la coalescence. La ligne 6 de l'algorithme 4.2.1 illustre le fait qu'on choisit aléatoirement deux individus, puis à la ligne 7 on calcule le résultat de la coalescence. Ensuite on ajoute la coalescence à l'arbre avant de mettre à jour les nœuds actifs.

4.2.1 Théorie de la coalescence avec mutations

Il existe plusieurs façons d'intégrer des évènements de mutation dans un arbre généré à partir de la théorie de la coalescence. La méthode utilisée dépend fortement du but que l'on cherche à accomplir ainsi que des hypothèses faites par le modèle qu'on utilise. Par exemple, si l'on cherche à simuler des arbres seulement d'une manière prospective on peut se servir des temps générés sur chaque branche de l'arbre pour apposer des mutations selon un taux de mutation μ. En utilisant le modèle des sites infinis, on peut apposer les mutations d'une manière uniforme sur une certaine longueur de séquences. Si notre modèle n'est pas basé sur l'hypo-

Algorithme 4.2.1 Algorithme permettant de construire un arbre phylogénique en utilisant la théorie de la coalescence.

1: **fonction** CONSTRUIREPHYLOGÉNIEAVECCOALESCENCE(n, N : entier) : arbre
2: Arbre *arbreResultat* ← un arbre vide()
3: Ensemble $noeudsActifs$ ← n nœuds
4: **tant que** $|noeudsActifs| > 1$ **faire**
5: réel $T \leftarrow \sim \exp(\frac{n(n-1)}{4N})$
6: $e_1, e_2 \leftarrow noeudsActifs[i, j]$
7: $e_{1+2} \leftarrow e_1 + e_2$
8: $noeudsActifs \leftarrow noeudsActifs \setminus \{e_1, e_2\} \cup \{e_{1+2}\}$
9: arbreResultat ← arbreResultat + $\mathcal{C}^T_{e_1,e_2}$
10: **fin tant que**
11: **retourner** *arbreResultat*
12: **fin fonction**

thèse de l'homoplasie, on peut apposer les mutations dans un modèle de sites finis et contraindre l'apparition de nouvelles mutations. Une des manières que nous utiliserons plus loin consiste à générer les mutations de manière indépendante aux coalescences. On peut alors choisir un des deux évènements proportionnellement à μ, n et N. Soit $\bar{H}$ le ratio des évènements de mutation divisé par la probabilité d'une coalescence et d'une mutation combinée. On obtient alors que :

$$\begin{aligned} \bar{H} &= \frac{\mu}{\mu + \frac{1}{2N}} \\ &= \frac{\mu}{\left(\frac{2N\mu + 1}{2N}\right)} \\ &= \frac{2N\mu}{2N\mu + 1}. \end{aligned} \tag{4.13}$$

Cette mesure quantifie l'intensité à laquelle les individus divergeront pendant leur évolution et nous nous servons d'équations semblables lors de la construction

d'ARG tel que présenté dans les chapitres suivants. Une des autres manières possibles pour intégrer les mutations consiste à construire une distribution exponentielle pour le taux de mutation entre chaque génération. Posons μ comme étant le taux de mutation par individu par génération. Le temps d'attente avant un évènement de mutation aura une distribution exponentielle avec comme paramètre $\lambda_{\mathcal{M}} = 2Nn\mu$. Comme le temps d'attente avant une coalescence est $\lambda_{\mathcal{C}}$, le temps d'attente avant une coalescence ou une mutation aura une distribution exponentielle de paramètre $\lambda_{\mathcal{C}} + \lambda_{\mathcal{M}}$. Pour construire une phylogénie, il suffit de simuler un temps d'attente et de choisir proportionnellement s'il s'agit d'un évènement de coalescence ou d'une mutation. Ensuite, selon le cas, on procède aux opérations selon le modèle établi.

CHAPITRE V

ALGORITHMES DE CONSTRUCTION

Dans ce chapitre, nous présentons en détails deux heuristiques de constructions d'ARGs existantes, soit Margarita (Minichiello et Durbin, 2006) et ARG4WG (Nguyen *et al.*, 2017). Ensuite nous détaillons une nouvelle heuristique que nous avons développée qui tire à profit les stratégies computationnelles que nous avons présentées au chapitre 3. De plus, nous décrivons sommairement une distribution statistique proposée par Fearnhead et Donnelly (Fearnhead et Donnelly, 2001) sur l'espace des graphes de recombinaison. Ceci permet de pondérer en quelque sorte tous les ARGs générés par différentes méthodes et ainsi améliorer la précision des résultats souhaités en composant plus efficacement les statistiques d'un ensemble de graphes. Toutefois, procéder de cette façon est computationnellement ardu [1] et il n'est pas toujours évident que les résultats seront meilleurs selon le temps de calcul imparti. Avant de plonger plus en détails voici quelques notations et définitions que nous utilisons dans ce chapitre et les suivants.

---

1. *«The method uses a computationally intensive statistical procedure (importance sampling) to calculate the likelihood under a coalescent-based model.»*, (Fearnhead et Donnelly, 2001, p.1 (abstract))

5.1 Notations des opérations

Les séquences génétiques seront formées à l'aide de l'alphabet $\mathcal{A} \equiv \{0, 1, ?\} = \{\square, \blacksquare, \boxtimes\}$. L'ensemble de départ H_0 qui renferme les données initiales contenant n séquences sera dénoté de la manière suivante : $|H_0| = |\{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n\}| = n$. Si les séquences sont de longueur l, alors on a que $\mathcal{S}_i[j] \in \mathcal{A} \quad \forall i \in [1, n],\ \forall j \in [1, l]$.

5.1.1 Les coalescences

Un évènement de coalescence entre deux séquences $\mathcal{S}_i$ et $\mathcal{S}_i'$ à la génération H_k engendre une nouvelle séquence $\mathcal{S}_i''$ à la génération H_{k+1}. La nouvelle séquence générée par la coalescence est notée : $\mathcal{C}(\mathcal{S}_i, \mathcal{S}_{i'}) = \mathcal{S}_i''$. Enfin, la génération qui suit cet évènement est produite de la manière suivante :

$$\mathcal{H}_{k+1} = \mathcal{C}_{i,i'}^{i''}(\mathcal{H}_k) = (\mathcal{H}_k \setminus \{\mathcal{S}_i, \mathcal{S}_{i'}\}) \cup \{\mathcal{S}_i''\}, \tag{5.1}$$

où $\mathcal{S}_i$ et $\mathcal{S}_{i'} \in H_k$ et $\mathcal{S}_i \equiv \mathcal{S}_{i'}$ [1]. La séquence $\mathcal{S}_{i''}$ est calculée de la même façon qu'à la section 3.2.2.2.

5.1.2 Les mutations

Les mutations ne modifient pas la cardinalité de l'ensemble des séquences de la génération d'origine. Toutefois, la nouvelle séquence aura un marqueur non mutant au lieu d'un marqueur primitif. Donc, soit $\mathcal{S}_i \in H_k$ où $\mathcal{S}_i[j] = \blacksquare$ et $p \in [1, l]$, alors

1. Voir l'équation 3.5 pour la définition.

l'opération de mutation sera de la forme :

$$\mathcal{S}_i' = \mathcal{M}(\mathcal{S}_i, p) = \begin{cases} \mathcal{S}_i[p] & \forall\, j \neq p \text{ où } j \in [1, l], \\ \neg\mathcal{S}_i[p] = \square & \text{lorsque } j = p. \end{cases} \tag{5.2}$$

La génération qui suit une mutation est de la forme :

$$\mathcal{H}_{k+1} = \mathcal{M}_i^p(\mathcal{H}_k) = (\mathcal{H}_k \setminus \{\mathcal{S}_i, \}) \cup \{\mathcal{S}_i'\}, \tag{5.3}$$

5.1.3 Les recombinaisons

Enfin, un évènement de recombinaison scinde en deux parties une séquence de la génération H_k et ajoute deux nouvelles séquences à la génération H_{k+1}. Formellement, soit $\mathcal{S}_i \in \mathcal{H}_k$ que l'on recombine au point p, alors les deux séquences engendrées $\mathcal{S}_i'$ et $\mathcal{S}_i''$ par la recombinaison $\mathcal{R}(\mathbf{S}_i, p) = \{\mathbf{S}_i', \mathbf{S}_i''\}$ sont définies de la manière suivante :

$$\mathcal{S}_i'[j] = \begin{cases} \mathcal{S}_i[j]\ \forall\, j < p, \\ \boxtimes\ \forall j\ \geq p, \end{cases} \qquad \mathcal{S}_i''[j] = \begin{cases} \boxtimes\ \forall j\ < p, \\ \mathcal{S}_i[j]\ \forall\, j \geq p, \end{cases} \tag{5.4}$$

où $p \in [1, l-1]$ et $j \in [1, l]$. Notons que la position p correspond à tous les espaces possibles entre deux marqueurs, soit le point de recombinaison entre les *loci* p et $p+1$. Toutefois, selon le modèle employé, certains articles choisissent ce point sur une échelle continue, mais cela n'impacte pas directement le fonctionnement des algorithmes discutés dans les sections suivantes. Donc, l'opération de recombinaison sur la génération k peut être définie de la manière suivante :

$$\mathcal{H}_{k+1} = \mathcal{R}_i^p(\mathcal{H}_k) = (\mathcal{H}_k \setminus \{\mathcal{S}_i, \}) \cup \{\mathcal{S}_i', \mathcal{S}_i''\}, \tag{5.5}$$

5.2 Caractéristiques des ARGs

Soient c le nombre total d'évènements de coalescence, r le nombre total d'évènements de recombinaison. Alors m le nombre total d'évènements de mutation peut être calculé de la manière suivante :

$$m = \sum_{j=0}^{l} \delta^{\blacksquare}_{\mathcal{S}_i[j]}, \tag{5.6}$$

où δ est le symbole de Kronecker, c'est-à-dire :

$$\delta^{\alpha}_{\mathcal{S}_i[j]} = \begin{cases} 1 & \text{si } \forall i \in [1, |\mathcal{H}_0|] \; \exists i \mid \mathcal{S}_i[j] = \alpha \\ 0 & \text{sinon.} \end{cases}$$

Avec ces définitions on peut démontrer que le nombre total de nœuds contenus dans un ARG est donné par l'équation suivante :

$$|\mathcal{G}(\mathcal{H}_t)| = |\bigcup_{i=1}^{\tau} H_{i-1} \Delta H_i| = m + c + 2r + |H_0|. \tag{5.7}$$

En effet, tous les évènements de mutation et de coalescence ajoutent un nouveau nœud dans le graphe tandis que les recombinaisons en introduisent deux. De plus, le nombre de feuilles du graphe correspond par définition à $|H_0|$. On a aussi que :

$$|H_\tau| = |H_0| + r - c = 1. \tag{5.8}$$

Cette équation souligne le fait que la taille des ARGs est directement proportionnelle au nombre de recombinaisons. Puisque les modèles utilisés pour construire des ARGs ne font pas l'hypothèse de l'homoplasie (aucune mutation récurrente), les évènements de recombinaison sont inévitables si nous voulons parvenir à un

ancêtre commun. Le test des quatre gamètes [1] permet de se convaincre facilement de ce fait. En effet, il suffit de tenter de construire un ARG sans recombinaison avec $H_0 = \{\{0,0\},\{0,1\},\{1,0\},\{1,1\}\}$ pour s'apercevoir que c'est impossible. Le nombre de recombinaisons détermine donc directement la taille des ARGs. Nous utilisons ce type d'équation pour valider, entre autres, que les ARGs produits à l'aide des algorithmes que nous décrivons dans les sections suivantes ont produit des graphes cohérents. À ce jour, il n'existe aucun algorithme connu qui permet de construire un ARG avec le nombre minimal de recombinaisons nécessaire pour atteindre l'ancêtre commun. On ne connait pas non plus d'expression analytique qui calcule ce minimum. Actuellement, seulement des bornes sur ce nombre sont connues (Song et Hein, 2005, Myers et Griffiths, 2003).

5.3 Heuristiques

Les méthodes heuristiques présentées dans cette section sont essentiellement des algorithmes gloutons, ou «*greedy algorithm*» en anglais, qui permettent de générer rapidement des ARGs sans qu'ils soient optimaux.

5.3.1 Margarita

Proposée par Mark J. Minichiello et Richard Durbin en 2006, l'approche Margarita (Minichiello et Durbin, 2006) permet de générer un ARG simple efficacement. Voici les principales règles de l'heuristique :

1. une recombinaison survient si et seulement si aucune mutation ni coalescence n'est possible ;

1. `https://en.wikipedia.org/wiki/Four-gamete_test`

2. si plusieurs mutations et/ou coalescences sont possibles, l'évènement est choisi de manière aléatoire ;
3. une coalescence est permise si et seulement s'il existe au moins un *locus* ancestral commun (voir l'équation 3.5 de la coalescence stricte de la section 3.2.2.1) ;
4. dans le cas où une recombinaison est nécessaire, soit on effectue une recombinaison d'une manière aléatoire selon une probabilité p, ou on choisit une recombinaison qui maximise la distance entre deux marqueurs dont tous les SNPs sont compatibles ;
5. la coalescence qui suit une recombinaison non aléatoire doit faire coalescer le segment commun de séquences qui a été maximisé.

Pour mieux expliquer la quatrième règle, nous devons définir quelques notations supplémentaires. On dit que deux *loci* sont compatibles et on écrit alors $\mathcal{S}_a[j] \sim \mathcal{S}_b[j]$ s'ils ont le même état allélique, c'est-à-dire :

$$\mathcal{S}_a[j] \sim \mathcal{S}_b[j] \Longleftrightarrow \mathcal{S}_a[j] = \mathcal{S}_b[j] \text{ ou } \mathcal{S}_a[j] = \boxtimes \text{ ou } \mathcal{S}_b[j] = \boxtimes.$$

On dit qu'il existe un segment commun entre les marqueurs α et β, si et seulement si $\mathcal{S}_a[j] = \mathcal{S}_b[j] \; \forall \; \alpha \leq j \leq \beta$ et qu'il existe au moins un j tel que $\mathcal{S}_a[j] = \mathcal{S}_b[j] \neq \boxtimes$.

L'algorithme de Margarita cherche à trouver le plus long des segments communs entre les paires de séquences présentes à la génération traitée. On dit que le segment est maximal entre deux séquences lorsque $\beta - \alpha$ est maximisé. Notons que si nous avons un segment maximal noté $\{\mathcal{S}_1, \mathcal{S}_2\}[\alpha, \beta]$ et que $\alpha > 1$, alors clairement $\mathcal{S}_1[\alpha - 1] \neq \mathcal{S}_2[\alpha - 1]$ et tous les deux sont ancestraux. En effet, dans les cas contraires il suffirait d'ajouter le *locus* au segment commun, car il est compatible, mais comme le segment est maximal nous avons une contradiction des hypothèses

de départ. On trouve un raisonnement similaire dans le cas où $\beta < l$. Ce sont précisément ces sites que l'on cherche à identifier pour délimiter nos segments communs maximaux possibles dans le tableau 3.4. De plus, il est crucial que les recombinaisons n'incorporent jamais de séquence ayant uniquement du matériel non ancestral. En effet, dans le cas où les coalescences ne peuvent être effectuées que sur des séquences ayant au moins un *locus* ancestral commun, une recombinaison qui introduirait une telle séquence ne serait jamais coalescée. Par conséquent, si le critère d'arrêt de l'algorithme est l'obtention d'une génération de taille un (ce qui est souvent le cas), l'algorithme ne terminera jamais son exécution. Une fois les α et β maximaux trouvés, on appose une recombinaison entre les marqueurs $\alpha - 1, \alpha$ si $\alpha > 1$ et une recombinaison entre les marqueurs $\beta, \beta + 1$ si $\beta < l$. Si le segment maximal débute ou se termine à la fin des séquences, une seule recombinaison est nécessaire. Sinon deux recombinaisons doivent être effectuées.

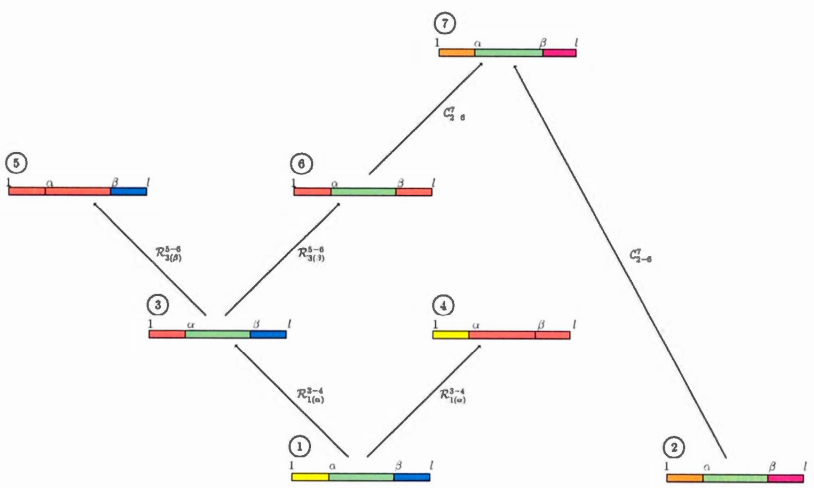


Figure 5.1 Recombinaison de type Magarita. Le matériel en rouge correspond au matériel non ancestral qui est introduit lors des recombinaisons. Les segments de séquences ayant la même couleur sont dits compatibles. On suppose que la distance entre les marqueurs α et β est maximale pour les séquences 1 et 2. Remarquons que $|H_0| = |\{S_1, S_2\}| < |\{S_4, S_5, S_7\}| = |H_3|$.

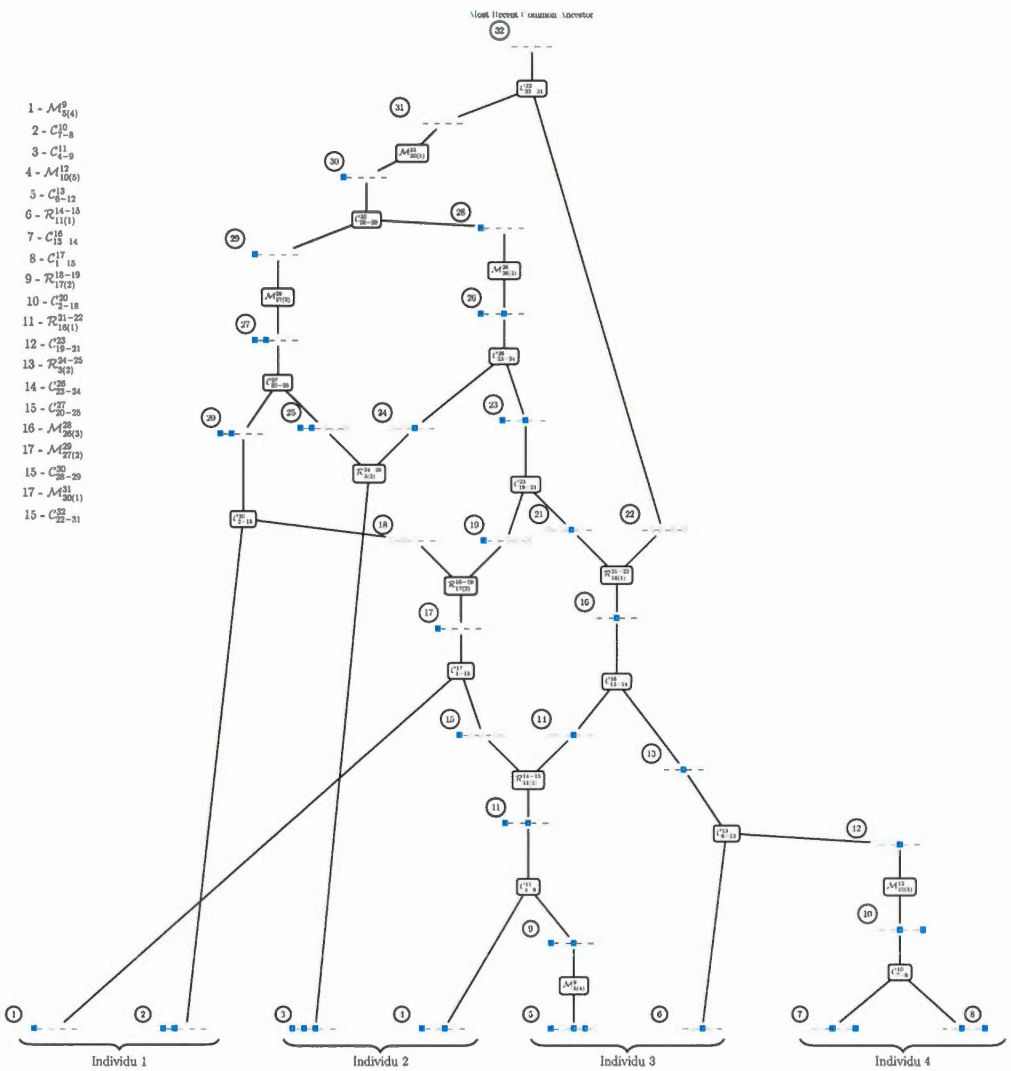


Figure 5.2 Graphe de recombinaison ancestral construit avec l'algorithme de Margarita. On voit qu'après les cinq premiers évènements (soit la génération : $H_5 = \{1, 2, 3, 11, 13\}$) aucune coalescence ou mutation n'est possible. La première coalescence qui suit la recombinaison est la fusion de deux segments de longueur 4, ce qui est le plus long segment commun possible. À la génération $H_8 = \{2, 3, 16, 17\}$, on voit qu'encore une fois une recombinaison est nécessaire pour poursuivre la construction du graphe vers le MRCA. Les segments maximaux ont une longueur de trois marqueurs au maximum, donc on effectue aléatoirement la recombinaison entre les marqueurs 2 et 3 sur le nœud 17. Une situation similaire survient lors des générations $H_{10} = \{3, 16, 19, 20\}$ et $H_{12} = \{3, 20, 22, 23\}$ avant d'avoir une recombinaison sur le nœud 3.

Lors de la construction de l'ARG contenu dans la figure 5.2, plusieurs recombinaisons sont possibles lorsqu'il existe plus d'un segment maximal de même longueur. On choisit alors uniformément un des évènements possibles parmi les segments maximaux possibles. Puisque les auteurs souhaitaient aussi que l'heuristique ne soit pas complètement déterministe, ils ont introduit une variable aléatoire qui change la manière dont un évènement de recombinaison est sélectionné comme discuté au début de cette section. Cela se reflète à ligne 8 de l'algorithme 5.3.1.

Algorithme 5.3.1 Algorithme permettant de construire un graphe de recombinaison ancestral selon l'algorithme de Margarita. Notons que les feuilles du graphe, notées $\mathcal{F}$, correspondent à l'échantillon de départ H_0.

1: **fonction** ConstruireARGSelonMargarita($\mathcal{F} : \{\mathcal{S} | \mathcal{S} \in \mathcal{A}^n\}$, $p : \in [0,1]$) : G : un ARG
2: $H_0 \leftarrow \mathcal{F}$
3: $G \leftarrow H_0$
4: **tant que** $|G.H_t| > 1$ **faire**
5: $G \leftarrow$ Générez toutes les mutations et les coalescences possibles aléatoirement.
6: **si** $G.H_t > 1$ **alors**
7: $va \leftarrow valeurRéelAléatoire[0,1]$
8: **si** $va \leq p$ **alors**
9: $G \leftarrow$Générez une recombinaison avec segment maximal.
10: **sinon**
11: $G \leftarrow$ Générez une recombinaison aléatoire.
12: **fin si**
13: **fin si**
14: **fin tant que**
15: **retourner** G
16: **fin fonction**

5.3.2 ARG4WG

L'algorithme ARG4WG (Nguyen *et al.*, 2017) est semblable à celui de Margarita. La différence majeure provient de la façon dont les recombinaisons sont choisies. Premièrement, l'algorithme ne fait pas de recombinaison aléatoire. Deuxièmement, les segments communs maximaux doivent inévitablement commencer par le premier *locus* ou terminer par le dernier *locus*. L'avantage principal de choisir les recombinaisons de cette manière est que toutes les générations consécutives (en omettant les générations recombinantes) ne contiennent jamais plus de séquences que la précédente. Cela est dû au fait qu'une seule recombinaison est nécessaire pour trouver et coalescer le segment maximal. Dans la figure 5.1 on peut voir qu'après un évènement de recombinaison, il y a un nœud de plus qu'au départ. Donc si plusieurs évènements de recombinaison nécessitant deux scissions pour coalescer un segment surviennent, la taille des générations produites peut croître rapidement et ralentir considérablement le calcul de l'ARG. Cela peut être évité avec cet algorithme, mais en contrepartie cela risque d'introduire un biais dans la construction des ARGs. En effet, si les marqueurs mutants ne sont pas distribués uniformément, comme une asymétrie de la densité des marqueurs mutants par exemple, l'algorithme aura alors tendance à trouver beaucoup plus rapidement un ancêtre commun pour les sections avec un faible taux de marqueurs mutants.

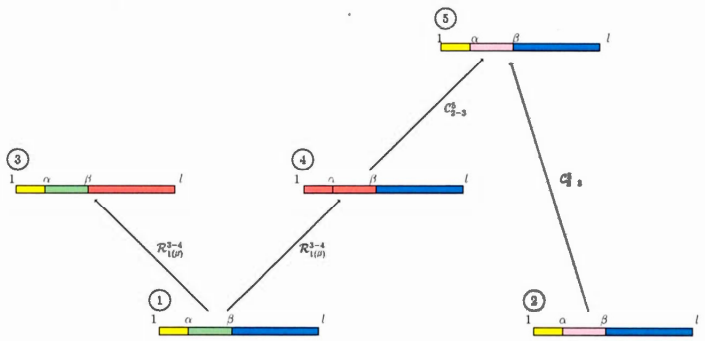


Figure 5.3 Recombinaison de type ARG4WG. Les séquences 1 et 2 comportent des segments compatibles aux extrémités. Si le segment en bleu est le plus long de toute une génération, alors ce segment sera coalescé immédiatement à la génération suivante. On remarque que : $|H_0| = |\{1,2\}| = |\{3,5\}| = |H_2| = 2$.

Les auteurs de cette heuristique ont comparé les temps d'exécution avec Margarita pour des données identiques et ils obtiennent une augmentation significative des performances. Pour expliquer ces gains, les auteurs présentent un lemme qui démontre que le nombre minimal de recombinaisons nécessaire pour intégrer complètement une séquence dans une génération est obtenu en choisissant systématiquement le plus long segment successivement à partir d'une même extrémité. Toutefois, leur algorithme n'utilise pas cette approche et les coalescences effectuées après chacune des recombinaisons mènent potentiellement à des mutations qui influencent le nombre de coalescences possible. Donc le gain de performance observé ne peut pas provenir uniquement de la stratégie de sélection des recombinaisons. Il serait intéressant de comparer l'efficacité intrinsèque des implémentations des deux algorithmes.

5.3.3 Manhattan

Dans cette section, nous proposons une nouvelle heuristique que nous avons développée. Comme la quantité des données initiales influence largement la qualité des ARGs obtenues, il est important de trouver des algorithmes qui permettent de traiter un maximum de données possible. Contrairement aux heuristiques précédentes, Manhattan priorise donc la réduction du nombre de marqueurs mutants présents d'une génération à l'autre. Cela permet de repérer des mutations non récurrentes plus rapidement et par le fait même, d'augmenter le nombre de coalescences potentielles.

Il y a également quelques arguments biologiques pour procéder de cette façon. Considérons la séquence possédant le plus grand nombre de marqueurs mutants à une génération quelconque. Alors il est plus probable que cette séquence ait elle-même hérité de segments ayant une grande proportion de *loci* mutants. Le plus rapidement ces segments de séquences seront coalescés, le plus rapidement nous parviendrons à trouver un MRCA. D'autre part, comme nous basons nos recombinaisons sur le nombre de marqueurs mutants et non sur la longueur des segments, l'algorithme ne peut pas introduire de biais provenant de la manière dont les SNPs ont été sélectionnés. Les segments de séquences non maximaux générés avec Margarita et ARG4WG ne seront pas coalescés rapidement, car ils sont généralement le restant d'un long segment commun. Donc, les générations vont conserver beaucoup de petites séquences avant de les coalescer vers le haut du graphe.

L'algorithme est lui aussi similaire à Margarita lorsqu'une coalescence et/ou une mutation sont possibles. C'est-à-dire qu'on choisit uniformément parmi un des

évènements possibles. Dans le cas où une recombinaison est nécessaire, Manhattan trouve dans un premier temps la séquence qui possède le plus grand nombre de marqueurs mutants. Si plusieurs séquences possèdent le même nombre maximal de marqueurs mutants, on choisit alors une de ces séquences aléatoirement. Puis on trouve à partir du premier marqueur le plus long segment compatible avec une autre séquence de la génération comportant le plus grand nombre de marqueurs mutants. Après la coalescence de ces séquences, nous avons toujours une séquence additionnelle à laquelle du matériel non ancestral a été ajouté du premier marqueur jusqu'au point de recombinaison. On recommence alors le processus avec cette séquence jusqu'à ce que toute la séquence ait été coalescée. De cette manière, la quantité de matériel non ancestral introduit est limitée et le nombre de séquences entre les générations ne peut pas augmenter significativement. La figure 5.4 illustre que ceci n'est pas toujours possible et qu'il existe un cas particulier qu'il faut traiter distinctement.

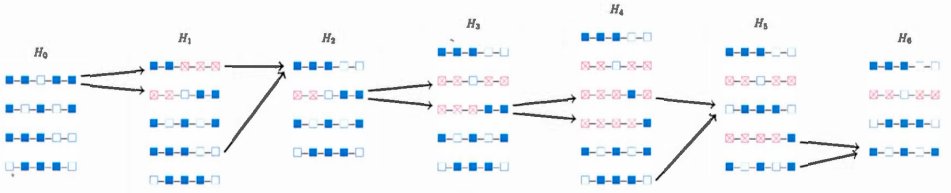


Figure 5.4 Illustration du comportement de l'algorithme de Manhattan lorsqu'aucune coalescence ni mutation n'est possible. La séquence choisie sur laquelle les recombinaisons seront effectuées est celle qui comporte le plus grand nombre de marqueurs mutants génétiques, soit quatre. Puis à chaque génération suivante, on coalesce le plus long segment possible toujours sur cette même séquence. À la génération H_3 on génère une séquence comportant un seul marqueur primitif, car toutes les autres séquences possèdent un marqueur mutant à ce locus.

Le nombre de séquences ajoutées après que l'algorithme de Manhattan ait effectué ses recombinaisons est égal au nombre de segments non contigus où des marqueurs

non ancestraux sont présents uniquement sur la séquence en traitement. Ces segments pourront seulement être coalescés vers le haut du graphe près du MRCA. Ce qui est cohérent, car ces *loci* ont une forte proportion de marqueurs mutants et donc ces séquences n'ont vraisemblablement pas contribué significativement au génotype de la population de départ. Elles pourraient, à la limite, être effacées des générations subséquentes, car il est peu probable qu'elles affectent d'une manière tangible la topologie des graphes. Sans compter le fait qu'elles détiennent une forte proportion de marqueurs non ancestraux. On peut voir que l'ARG construit avec l'heuristique Manhattan à la figure 5.5 possède les mêmes données présentes dans l'ARG de la figure 5.2 et on parvient au MRCA avec deux recombinaisons de moins, soit quatre contre deux.

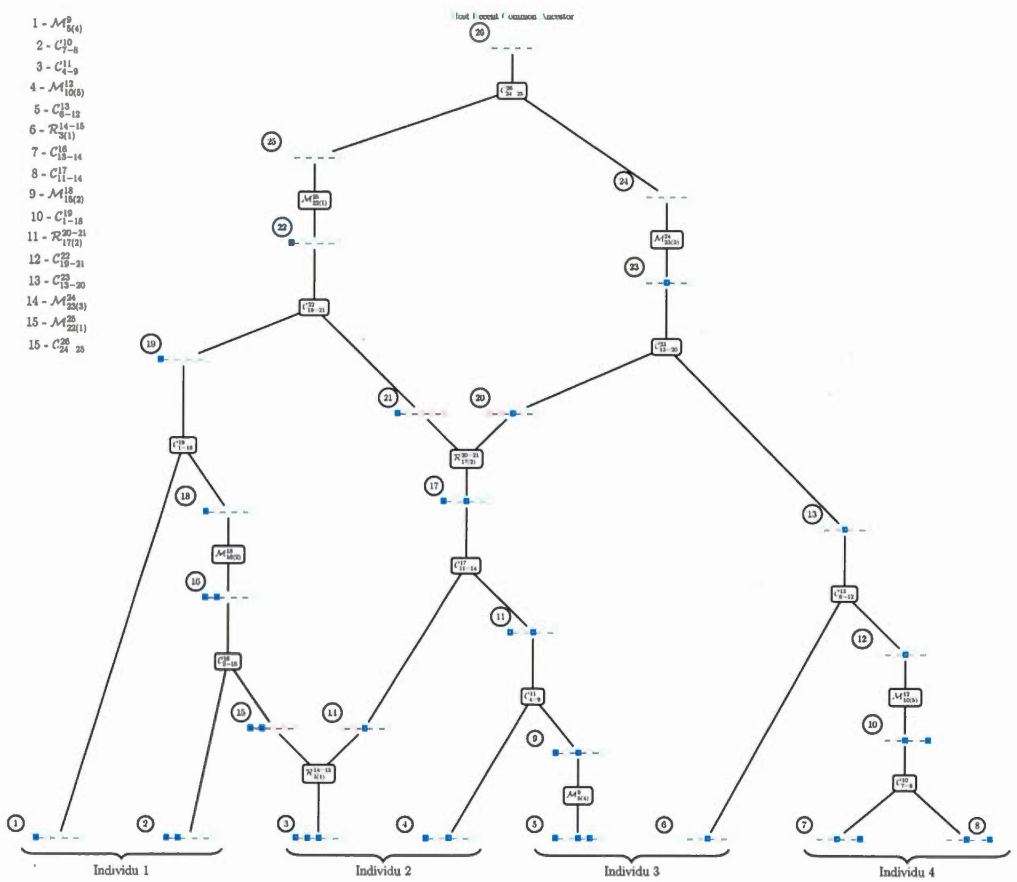


Figure 5.5 Graphe de recombinaison ancestral construit avec l'algorithme de Manhattan. On peut voir que les segments de séquences générés sont immédiatement coalescés dans les générations suivantes. De plus, on remarque que la recombinaison effectuée à la génération $H_5 = \{1, 2, 3, 11, 13\}$ est pratiquée sur la séquence qui possède le plus de marqueurs mutants. Idem pour la génération $H_{10} = \{13, 17, 19\}$.

On peut vérifier que l'équation (5.7) est satisfaite pour le graphe de la figure 5.5 :

$$\begin{aligned} |\mathcal{G}(\mathcal{H}_\prime)| &= |\bigcup_{i=1}^{\tau} H_{i-1} \Delta H_i| \\ &= m + c + 2r + |H_0| \\ &= 5 + 9 + 4 + 8 \\ &= 26. \end{aligned}$$

Nous détaillons plus longuement l'implémentation que nous avons mise au point pour cet algorithme ainsi que quelques résultats de temps d'exécution dans les chapitres suivants.

5.4 Distribution statistique

Nous survolons dans cette section une méthode qui génère des ARGs selon une distribution, plutôt que certains algorithmes qui produisent des ARGs aléatoirement seulement. Nous utiliserons une technique d'échantillonnage préférentiel qui pondère les graphes compatibles avec un ensemble de séquences initial. Dans un premier temps, nous présentons un article reconnu qui suggère une distribution optimale. Puis nous décrirons une distribution alternative que nous avons mise au point.

5.4.1 Distribution optimale

L'article de Fearnhead & Donnelly cherche principalement à estimer le taux de mutation et de recombinaison à l'aide d'un estimateur de vraisemblance qui utilise un échantillonnage préférentiel d'un ensemble d'ARGs générés. Pour ce faire, ils ont développé une distribution qui pondère chaque graphe dans le but de construire

une surface estimant ces deux paramètres.

5.4.1.1 Stratégie

Soit $\mathscr{G}$ l'espace qui contient tous les graphes de recombinaison ancestraux, E un échantillon de n séquences observées. Supposons maintenant qu'on cherche à estimer un paramètre φ, on obtient alors la fonction de vraisemblance suivante :

$$\mathcal{L}(\varphi) = p(E \mid \varphi) = \int p(E \mid A, \varphi) p(A \mid \varphi) \, \mathrm{d}A, \tag{5.9}$$

où $A \in \mathscr{G}$. La probabilité de produire un ARG aléatoire cohérent avec un échantillon quelconque est pratiquement nulle. Par conséquent, presque tous les termes de cette fonction de vraisemblance n'auront aucune contribution, ce qui rend son calcul pratiquement impossible. De plus, comme le temps nécessaire pour produire un seul ARG est non négligeable, il est crucial de réduire au maximum le nombre d'ARG qu'on doit générer pour obtenir notre vraisemblance. Les ARGs générés doivent donc être le plus proche possible du parcours évolutif de l'ensemble de départ des séquences si on désire obtenir de bons résultats. Toutefois, si $A \in \mathscr{G}(\varphi)$ où $\mathscr{G}(\varphi)$ correspond à l'ensemble des graphes possibles ayant φ comme paramètre, alors le terme $p(E \mid A, \varphi)$ devient une fonction indicatrice valant 1 si l'échantillon est cohérent et 0 dans le cas contraire. Donc, si on introduit un échantillonnage préférentiel sur l'espace des ARGs, on réduira énormément le nombre de calculs à effectuer. En posant $q(G)$ comme une fonction de densité définie sur le domaine $\mathcal{G} = \mathscr{G}(\varphi, E)$ qui correspond à l'ensemble des graphes cohérents avec nos données initiales E, on obtiendra une fonction dont le calcul est plus réaliste. Nous pouvons maintenant réécrire la vraisemblance de cette manière :

$$\mathcal{L}(\varphi) = \int_{\mathcal{G}} \frac{p(E \mid G, \varphi) p(G \mid \varphi)}{q(G)} q(G)\, \mathrm{d}G \tag{5.10}$$

$$= \int_{\mathcal{G}} \frac{p(G|\varphi)}{q(G)} q(G)\, \mathrm{d}G \tag{5.11}$$

$$= \mathbf{E}[\omega(\mathcal{G})] \tag{5.12}$$

où $G \in \mathcal{G}$ et $\omega(G) = p(G|\varphi)/q(G)$. Le passage de l'équation (5.10) à (5.11) s'explique par le fait que comme $\mathcal{G} \subset \mathscr{G}(\varphi, E)$, alors le terme $p(E \mid G, \varphi) = 1$ pour tout $G \in \mathcal{G}$ par définition. Pour le passage de l'équation (5.11) à (5.12) cela provient directement de la définition de l'espérance mathématique et le terme $\omega(\mathcal{G})$ se nomme le poids de l'échantillonnage préférentiel. Posons maintenant $V = \mathbf{E}[\omega(\mathcal{G})]$, alors cela peut être estimé par :

$$\widetilde{V_M} = \frac{1}{M} \sum_{i=1}^{M} \omega(G_i),$$

où $\{G_1, ..., G_M\}$ est un échantillon i.i.d. inclus dans $\mathcal{G}$ généré selon la densité q. Ainsi nous pouvons réécrire la vraisemblance de la manière suivante :

$$\mathcal{L}(\varphi) = \int_{\mathcal{G}} \frac{p(G|\varphi)}{q(G)} q(G)\, \mathrm{d}G \approx \frac{1}{M} \sum_{i=1}^{M} \frac{p(G_i|\varphi)}{q(G_i)} \tag{5.13}$$

où les G_i appartiennent à $\mathcal{G}$. Ceci étant défini, nous devons maintenant trouver une façon de calculer les termes de type $p(G_i)$ et $q(G_i)$. Premièrement, puisqu'on construit les ARGs selon une suite d'états (les générations) conformément à un processus de Markov (les évènements subséquents sont dépendants uniquement de la génération actuelle), il est naturel de définir les termes (G_i) comme le produit des probabilités de transitions. C'est-à-dire :

$$p(G_i) = \prod_{i=0}^{\tau - 1} p(H_i | H_{i+1}), \tag{5.14}$$

où $H_0 = E$ et H_τ correspond à la génération qui contient seulement le plus récent ancêtre commun (MRCA). Notons que les probabilité de transition vers le futur

sont connues, mais pas celles vers le passé tel que $p(H_{i+1}|H_i)$. Notons que nous pouvons omettre le conditionnement sur le paramètre φ car cela ne changerait pas le calcul de $p(G_i)$. À partir de ce point, la méthode employée par Fearnhead diverge considérablement des algorithmes que nous avons présentés. Entre autres, en plus des évènements possibles que nous avons décrits, les auteurs ajoutent un évènement d'imputation, dans lequel on assigne aux *loci* non ancestraux un état primitif ou mutant en fonction de la proportion des états observés sur les autres séquences à chacun des *loci* en question. Donc nous survolerons seulement les éléments essentiels de la densité proposée.

5.4.2 Fonction de densité

La densité optimale prend la forme suivante :

$$q(H_{i+1}|H_i) = p(H_i|H_{i+1})\frac{\pi(H_{i+1})}{\pi(H_i)} \tag{5.15}$$

où $\pi(H_i)$ est la probabilité d'observer les haplotypes présents de la génération H_i. Similairement à l'équation (5.14) on peut obtenir une probabilité sur un ARG de la manière suivante :

$$p(G_i) = \prod_{i=0}^{\tau-1} q(H_{i+1}|H_i). \tag{5.16}$$

En substituant cette équation ainsi que l'équation (5.15) dans l'équation (5.13),

on obtient :

$$\begin{aligned}\mathcal{L}(\varphi) &\approx \frac{1}{M}\sum_{i=1}^{M}\frac{p(G_i|\varphi)}{q(G_i)} \\ &= \frac{1}{M}\sum_{i=1}^{M}\frac{\prod_{i=0}^{\tau-1}p(H_i|H_{i+1})}{\prod_{i=0}^{\tau-1}q(H_{i+1}|H_i)} \\ &= \frac{1}{M}\sum_{i=1}^{M}\frac{\prod_{i=0}^{\tau-1}p(H_i|H_{i+1})}{\prod_{i=0}^{\tau-1}p(H_i|H_{i+1})\frac{\pi(H_{i+1})}{\pi(H_i)}} \\ &= \frac{1}{M}\sum_{i=1}^{M}\prod_{i=0}^{\tau-1}\frac{\pi(H_i)}{\pi(H_{i+1})}.\end{aligned} \tag{5.17}$$

Pour calculer ce ratio présent à la dernière ligne, les auteurs remarquent que lorsqu'on passe d'une génération à l'autre, seulement une ou deux séquences sont remplacées. Ils définissent alors une nouvelle distribution conditionnelle. Cette distribution conditionne les nouvelles séquences ajoutées sachant les haplotypes contenus dans une génération H :

$$\pi(\alpha|H) = \frac{\pi(\{H,\alpha\})}{\pi(H)}. \tag{5.18}$$

Maintenant, supposons qu'une coalescence survient entre deux séquences totalement identiques α, la génération suivante H_{i+1} correspond à la génération courante à laquelle une séquence α a été retranchée : $H_{i+1} = H_i - \alpha$. On peut maintenant définir le ratio convoité de la manière suivante :

$$\begin{aligned}\frac{\pi(H_i)}{\pi(H_{i+1})} &= \frac{\pi(H_i)}{\pi(H_i-\alpha)} \\ &= \frac{\pi(H_i)}{\left(\frac{\pi(H_i-\alpha,\alpha)}{\pi(\alpha|H_i-\alpha)}\right)} \\ &= \frac{\pi(H_i)\pi(\alpha|H_i-\alpha)}{\pi(H_i-\alpha,\alpha)} \\ &= \pi(\alpha|H_i-\alpha).\end{aligned}$$

Toutefois, cette densité est inconnue et les auteurs suggèrent donc une approximation possible pour leur modèle. Par exemple, une approximation pour un modèle qui ne comporte pas de recombinaison à déjà été étudié auparavant (Stephens et Donnelly, 2000). Comme plusieurs transitions sont possibles, les auteurs choisissent au préalable une des séquences pour laquelle un évènement sera choisi aléatoirement. Ce qui diminue considérablement la complexité du calcul des ratios de transitions à effectuer. Comme ils cherchent à estimer deux paramètres, soit θ le taux de mutation et ρ le taux de recombinaison, les auteurs choisissent une séquence proportionnellement à : $n - 1 + a_i\theta + b_i\rho$ où n est le nombre de séquences, a_i est la proportion du matériel ancestral de la séquence S_i et b_i la distance totale entre le premier et le dernier marqueur ancestral. Une fois la séquence choisie, on effectue un des évènements possibles selon les probabilités suivantes :

1. une coalescence survient entre deux séquences identiques avec probabilité :

$$C_{\alpha,\alpha'}^{\alpha''} = \left(\frac{\mathcal{S}_\alpha - 1}{p(\alpha|H-\alpha)}\right),$$

   où $\mathcal{S}_\alpha$ est le nombre de séquences identique α à la génération H ;

2. une coalescence entre deux séquences non identiques α et β, ce qui produit une séquence γ avec probabilité :

$$C_{\alpha,\beta}^{\gamma} = \left(\frac{p(\gamma|H-\alpha-\beta)}{p(\alpha|H-\alpha)p(\beta|H-\alpha-\beta)}\right),$$

3. une mutation sur un *locus* j d'une séquence α produisant une nouvelle séquence β avec probabilité :

$$\mathcal{M}_{\alpha(j)}^{\beta} = p(\mathcal{S}_\beta[l]|\mathcal{S}_\alpha[l])\theta\left(\frac{p(\beta|H-\alpha)}{p(\alpha|H-\alpha)}\right),$$

   où $p(\mathcal{S}_\beta[l] \mid \mathcal{S}_\alpha[l])$ correspond au cas où un allèle $\mathcal{S}_\alpha[l]$ mute vers un allèle $\mathcal{S}_\beta[l]$ (voir la figure 1.1 ainsi que le tableau 1.3).

4. une recombinaison survient entre les marqueurs j et $j+1$ sur une séquence α ce qui produit deux séquences β et γ avec probabilité :

$$\mathcal{R}^{\beta,\gamma}_{\alpha(j)} = d\rho \left(\frac{p(\beta|H-\alpha)p(\gamma|H-\alpha+\beta)}{p(\alpha|H-\alpha)} \right),$$

où d est la distance entre les deux marqueurs qui sépare le point de recombinaison.

Notons que tous ces termes sont normalisés par une constante dans le but que la somme des évènements possibles soit égale à un. Nous ne détaillons pas le calcul des probabilités de type $p(\alpha|H)$, ce qui est fait dans l'article (voir (Fearnhead et Donnelly, 2001, APPENDIX A)), toutefois nous proposons une manière alternative de calculer la fonction de vraisemblance.

5.4.3 Densité alternative

Nous devons d'abord définir trois termes correspondant à la somme des événements possibles pour produire une génération H_{i+1} à partir de H_i. Soit $\sigma(H_i)$ le nombre total de coalescences possibles définies de la manière suivante :

$$\sigma(H_i) = |\{\mathcal{C}_{\mathcal{S}_i,\mathcal{S}_j} | \mathcal{S}_i \equiv \mathcal{S}_j,\ i \in [1,n],\ j > i, \mathcal{S}_i \in H_i\}|. \tag{5.19}$$

Ensuite, nous définissons de manière similaire le nombre de mutations possibles $\mu(H)$ à une génération. Ce qui correspond au nombre de sites singletons mutants de la manière suivante :

$$\mu(H_i) = |\{\mathcal{M}_{i(j)} | \forall i \in [1,n],\ \exists! \mathcal{S}_i[j] = \blacksquare\ \forall j \in [1,l]\}|. \tag{5.20}$$

Enfin, le nombre de recombinaisons ρ possibles est donné par la somme des distances entre les premiers et les derniers sites ancestraux pour une génération H_i. Nous pouvons définir de la manière suivante :

$$\rho(H_i) = |\{\mathcal{R}_{i(j)} | \forall i \in [1,n], j \in [l,r[\}|, \tag{5.21}$$

où l est la position du premier *locus* ancestral et r la position du dernier. Posons maintenant la densité des probabilités de transitions selon l'équation suivante :

$$\pi(H_{i+1}|H_i) = p(H_i|H_{i+1})(\sigma(H_i) + \mu(H_i) + \rho(H_i)). \tag{5.22}$$

On obtient donc la vraisemblance suivante :

$$\begin{aligned}
\mathcal{L}(\varphi) &\approx \frac{1}{M}\sum_{i=1}^{M}\frac{p(G_i|\varphi)}{q(G_i)} \\
&= \frac{1}{M}\sum_{i=1}^{M}\frac{\prod_{i=0}^{\tau-1} p(H_i|H_{i+1})}{\prod_{i=0}^{\tau-1}\pi(H_{i+1}|H_i)} \\
&= \frac{1}{M}\sum_{i=1}^{M}\frac{\prod_{i=0}^{\tau-1} p(H_i|H_{i+1})}{\prod_{i=0}^{\tau-1} p(H_i|H_{i+1})(\sigma(H_i) + \mu(H_i) + \rho(H_i))} \\
&= \frac{1}{M}\sum_{i=1}^{M}\prod_{i=0}^{\tau-1}\frac{p(H_i|H_{i+1})}{p(H_i|H_{i+1})(\sigma(H_i) + \mu(H_i) + \rho(H_i))} \\
&= \frac{1}{M}\sum_{i=1}^{M}\prod_{i=0}^{\tau-1}\frac{1}{(\sigma(H_i) + \mu(H_i) + \rho(H_i))}.
\end{aligned}$$

Toutefois, cette vraisemblance possède un défaut computationnel majeur. Comme les termes sont relativement petits ($\propto 1/(n \times l)$ où n est le nombre de séquences et l est la longueur des séquences) et qu'ils sont multipliés pour toutes les générations de chacun des graphes, le calcul de cette fraction en point flottant risque de dépasser la résolution maximale du type employé. Pour éviter ce problème, on peut prendre l'exponentielle du logarithme naturel et traiter le produit comme une somme. Il suffit par la suite d'ajuster la vraisemblance obtenue.

$$\begin{aligned}\mathcal{L}(\varphi) &\approx \frac{1}{M}\sum_{i=1}^{M}\left(\prod_{i=0}^{\tau-1}\frac{1}{(\sigma(H_i)+\mu(H_i)+\rho(H_i))}\right)\\ &= \frac{1}{M}\sum_{i=1}^{M}\exp\left(\ln\left(\prod_{i=0}^{\tau-1}\frac{1}{(\sigma(H_i)+\mu(H_i)+\rho(H_i))}\right)\right)\\ &= \frac{1}{M}\sum_{i=1}^{M}\exp\left(\sum_{i=0}^{\tau-1}\ln\left(\frac{1}{(\sigma(H_i)+\mu(H_i)+\rho(H_i))}\right)\right).\end{aligned}$$

Cette vraisemblance correspond à la moyenne des produits des probabilités de transitions pour un ensemble d'ARGs généré en fonction d'un paramètre que l'on souhaite tester. Intuitivement, si l'on choisit φ qui est éloigné de la valeur réelle, les ARGs seront en principe plus laborieux à construire. Ce qui donnera lieu à une inflation du nombre de générations nécessaire et la quantité totale d'évènements possibles va augmenter. Cela va se traduire par une probabilité plus faible pour cette valeur estimée. On voit aussi que l'évaluation de cette vraisemblance ne devrait plus causer de problème computationnel, car chaque terme reste contraint dans un domaine raisonnable par la fonction logarithmique.

CHAPITRE VI

IMPLÉMENTATION

Dans ce chapitre, nous présentons sommairement *FastARG* [1], un logiciel que nous avons conçu. Ce programme écrit principalement en *C++* permet de construire des graphes de recombinaison ancestraux selon les principes et fonctions que nous avons discutés dans les chapitres antérieurs. Nous détaillons plus particulièrement la façon dont les algorithmes et les opérations principales présentés auparavant sont mis en œuvre. Nous produisons aussi quelques graphiques illustrant la performance que nous obtenons pour ces fonctions.

6.1 Type abstrait de données

FastARG a été élaboré avec l'idée de dissimuler les opérations possibles sur des ARGs par des méthodes associées aux éléments essentiels qui composent les graphes. Donc nous avons élaboré plusieurs objets ainsi qu'une classe abstraite pour les différents types d'ARGs. Cette classe peut être héritée pour être spécialisée en fonction du type d'ARGs que l'on souhaite construire. De cette manière, il est aisé d'adapter la plupart des algorithmes de constructions d'ARGs. Cela permet aussi de comparer les divers algorithmes sur un même pied d'égalité. Voici donc

1. `https://bitbucket.org/fastarguqam/` - (13 septembre 2018)

l'équivalent d'un type abstrait de données (ADT) qui modélise le comportement sémantique des opérations que nous avons implémentées.

Tableau 6.1 Spécification sémantique de l'objet principal de FastARG, soit ARG qui englobe les aspects fondamentaux des graphes de recombinaison ancestraux.

Type :	ARG		
Utilise :	Booléen, Entier, Séquence		
Opérations :	creerARG :	Séquence	$\longrightarrow$ ARG
	genererUneCoalescence	Séquence $\times$ Séquence	$\longrightarrow$ ARG
	genererUneMutation	Séquence $\times$ Entier	$\longrightarrow$ ARG
	genererUneRecombinaison	Séquence $\times$ Entier	$\longrightarrow$ ARG

Sans faire une liste exhaustive des préconditions et axiomes comme une spécification formelle d'un type abstrait de données exigerait, on tient pour acquis que les séquences initiales ont la même longueur et que les opérations demandées sont valides par exemple. Dans la plupart des algorithmes, les évènements sont générés aléatoirement une fois que le type d'événements est choisi. Puisque notre logiciel possède tous les attributs et variables donnant accès à l'ensemble des opérations possibles, la cohérence de chacun des graphes construits est assurée par le fait qu'on utilise des services déjà testés en profondeur à chaque étape. En effet, nous avons dédié plus de trois mille lignes de code pour effectuer des tests unitaires sur l'ensemble des fonctions présentes dans *FastARG*. On peut voir à l'annexe C un extrait de ces tests.

6.2 Fonctionnement général de *FastARG*

FastARG utilise un fichier de configuration qui a recours au format «*JSON*» pour l'analyse syntaxique des paramètres utilisateur. Un deuxième fichier contenant les SNPs doit être fourni. Les résultats et statistiques sont produits dans un fichier texte. Nous présentons dans cette section seulement les fonctions essentielles à tous les algorithmes ainsi que certains détails de leur implémentation. Il est important de noter que lors de la construction d'un ARG, chaque fois qu'un évènement survient (voir section 5.1), *FastARG* recalcule l'ensemble des variables suivantes :

- le nombre total de coalescences possibles ;
- l'ensemble des paires de séquences pouvant coalescer ;
- le nombre de sites mutants pour chacun des *loci* ;
- le nombre total de sites mutants pour la génération courante ;
- le nombre de séquences dans la génération courante ;
- la topologie pour tous les nœuds internes pour tous les *loci* ;
- etc.

Bien que le calcul de toutes ces variables demande un temps considérable, elles permettent de générer des ARGs selon des modèles complexes. Le calcul de toutes les topologies réalisables est une des opérations les plus exigeantes, mais elle offre la possibilité de faire percoler efficacement les mutations de manière similaire à l'heuristique Margarita. De plus, avec toutes ces données disponibles directement à chaque génération, il devient trivial de créer diverses statistiques pour les graphes.

6.2.1 Implémentation du test de coalescence

6.2.1.1 Version sans contrainte

Nous avons décrit dans un chapitre précédent l'équation (3.3) qui vérifie la compatibilité entre deux séquences. Nous avons fait ceci dans le but de voir si une coalescence entre ces deux séquences est envisageable. Voici l'implémentation en *C++* de ce test :

```
1  bool SnipSequence::noConstraintCoalescence( const SnipSequence &firstSequence, const
       SnipSequence& secondSequence) {
2      std::valarray<DATA_TYPE> A = firstSequence.AncestralDataMask;
3      std::valarray<DATA_TYPE> B = firstSequence.snipsData;
4      std::valarray<DATA_TYPE> C = secondSequence.AncestralDataMask;
5      std::valarray<DATA_TYPE> D = secondSequence.snipsData;
6
7      size_t i = 0;
8      while (i < firstSequence.getnumberOfBlock() ){
9          if ( (~C[i])&(~A[i])&(((~D[i])&B[i])|(D[i] &(~B[i]))) ){
10             return false;
11         }
12         ++i;
13     }
14     return true;
15 }
```

Les séquences sont divisées en section de 32 ou 64 bits spécifiés selon l'expression «*DATA_TYPE*». Comme les séquences ne sont pas nécessairement des multiples de 32 ou 64, on emploie des *bits de bourrage* [sic] qui n'affectent pas le résultat du dernier bloc testé. Comme un seul site non compatible est nécessaire pour qu'une coalescence soit proscrite, on utilise une boucle de type infini qui termine l'exécution au premier bloc incompatible. Si tous les blocs passent le test, la fonction retourne vrai. La condition à ligne 9 correspond à un entier de 32 ou 64 bits qui est composé du résultat de la fonction binaire qui retourne la valeur 1 sur les positions incompatibles. Un seul bit distinct de «0» dans l'expansion binaire de l'entier suffit pour que celui-ci soit différent de zéro. Et comme en *C++* tout entier différent de zéro équivaut à la valeur vrai, on obtient donc que la fonction

retourne le booléen faux si une coalescence est impossible.

6.2.1.2 Parallélisation du test

Nous avons aussi tenté de produire une implémentation du test de coalescence sans contrainte en programmation parallèle. Nous avons utilisé la bibliothèque *Threading Building Blocks* (*TBB*) d'Intel pour implémenter notre parallélisation en mémoire partagée. Nous avons fait une réduction avec la fonction «*parallel_reduce*» sur des blocs à l'aide de variables de type «*blocked_range*» présent dans la bibliothèque TBB. Nous avons comparé les performances du mode séquentiel et pour des blocs assignés de manière statique et dynamique. Voici un extrait du code qui implémente la parallélisation dynamique de l'opération :

```
1  bool SnipSequence::operator ==(const SnipSequence& otherSnipSequence) const {
2      // Mode parallele avec taille des grains dynamique.
3      bool result = true;
4      tbb::blocked_range<unsigned int> r (0,SnipSequence::getnumberOfBlock());
5      result = tbb::parallel_reduce(
6          r,
7          result,
8          [&] (const tbb::blocked_range<unsigned int> r, bool in )
9          {
10             for (unsigned int i = r.begin() ; result && i < r.end()  ;++i){
11                 if ( (~C[i])&(~A[i])&(((~D[i])&B[i]) | (D[i] &(~B[i])))  ){
12                     result = false;
13                 }
14             }
15             return result;
16         },
17         std::plus<bool>()
18     );
19     return result;
20 }
```

Nous avons dans un premier temps effectué un test en augmentant progressivement la longueur des séquences pour comparer les temps d'exécution du mode séquentiel et parallèle. Nous pouvons constater à la figure 6.1 que la performance entre la fonction séquentielle et parallèle est très semblable. En fait, il y a une

explication simple pour cela. Premièrement, lors des premiers tests nous avons générés des séquences aléatoirement et nous obtenions systématiquement des temps beaucoup plus longs pour la fonction qui était parallélisée et ce peu importe la longueur des séquences. Il semblerait même y avoir une tendance négative, c'est-à-dire que plus les séquences étaient longues, plus la fonction parallèle était plus lente que la version séquentielle. Ceci est dû au fait que lorsque les séquences sont aléatoires, une incompatibilité devrait être vite repérée lors des premiers tests effectués. Mais comme la fonction parallèle possède un coût initial plus élevé pour mettre en place l'exécution, cela devenait très pénalisant et rendait à toute fin pratique les fonctions de test parallèle caduques. Pour obtenir ce résultat, nous avons donc choisi le meilleur cas possible pour l'algorithme parallèle. Donc les séquences sont presque toutes compatibles et les tests peuvent donc être effectués sur toute la longueur de la séquence. Même dans ce cas, des séquences avec 2^{29} SNPs ne suffisent pas pour obtenir un gain de performance. Donc, nous croyons que les tests de coalescence sont au moins aussi efficaces en mode séquentiel que parallèle.

La taille des paquets que chaque fil d'exécution doit traiter influe grandement sur l'accélération potentielle que l'on peut obtenir. Si la granularité est trop petite, la pénalité inhérente lors de l'instauration du calcul parallèle se fera sentir. Si au contraire la taille des grains est trop grande, il sera ardu de tirer avantage des multiples fils d'exécution. Normalement, l'assignation dynamique de la taille des grains devrait produire de meilleurs résultats, car il est optimisé par les concepteurs de la bibliothèque. Mais nous avons implémenté un mode statique pour mieux comprendre comment la taille des grains influencerait les résultats que nous avons obtenus.

Étant donné la difficulté d'implémentation du test des coalescences en programmation parallèle, nous n'avons pas choisi de poursuivre cette voie pour *FastARG*.

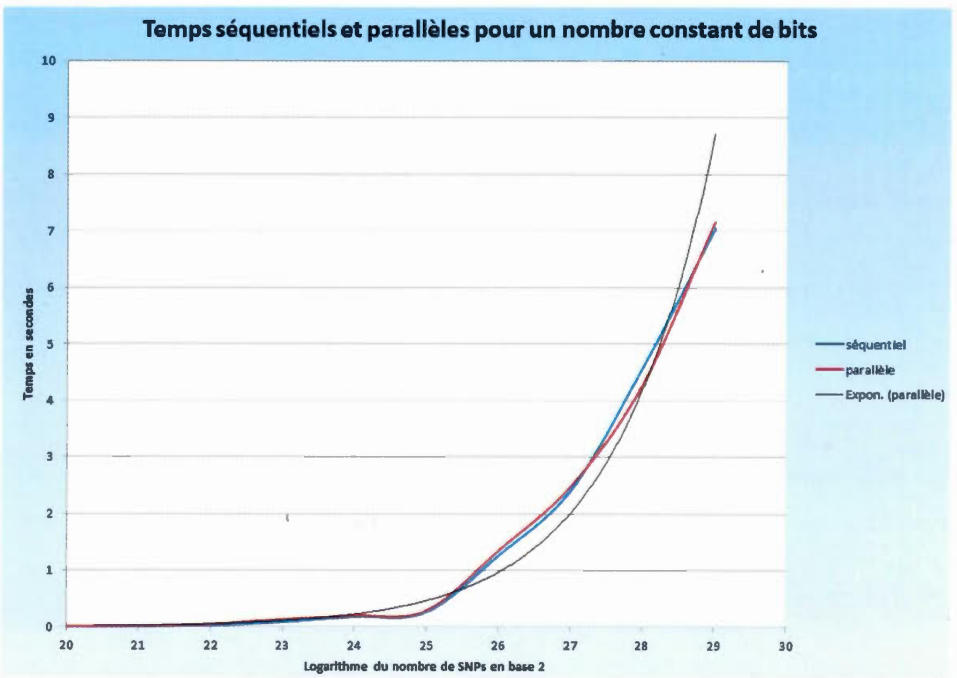


Figure 6.1 Temps d'exécution illustrant la performance de l'opérateur d'équivalence entre séquences en variant la longueur des séquences. Les fonctions séquentielles et parallèles conservent la taille et le nombre des séquences égales. Donc le nombre de bits est constant. Malgré plusieurs tests, nous avons rarement obtenu de meilleures performances de l'algorithme parallèle.

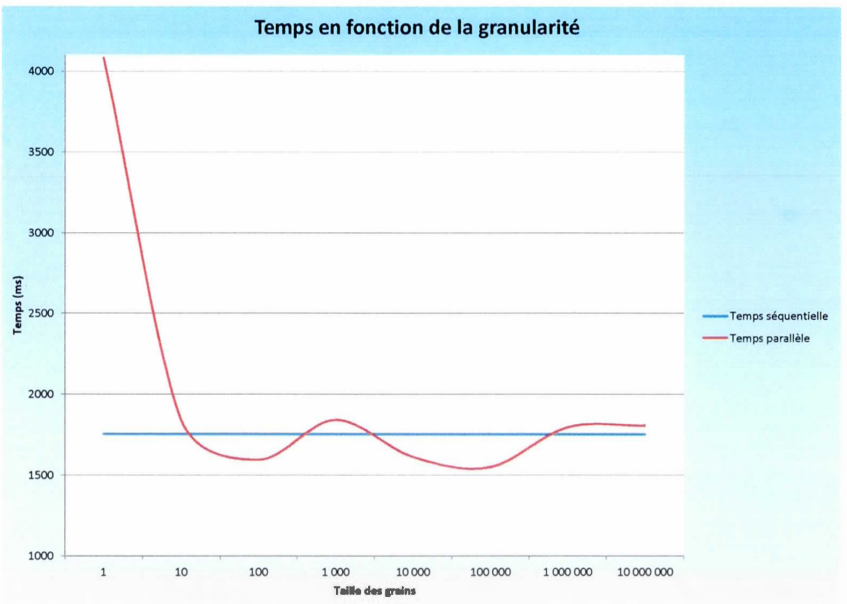


Figure 6.2 Données obtenues en testant si une coalescence est possible entre 50 séquences comportant un peu moins de 17 millions de SNPs. On peut constater que la fonction parallèle avec des grains de taille 1 est beaucoup plus lente que l'algorithme séquentiel. Ce qui est normal, toutefois nous ne voyons pas de gain substantiel, et ce peu importe la taille des grains. Il est possible que le compilateur optimise implicitement la fonction ce qui pourrait expliquer ce résultat.

6.2.2 Test de coalescence avec contrainte

Nous avons aussi implémenté le test qui vérifie qu'il existe au moins un *locus* ancestral commun entre les deux séquences testées. Nous avons décrit à l'équation (3.4) comment il est possible de procéder pour ce test avec des fonctions booléennes. Voici comment se traduit ce test supplémentaire en *C++* :

```
1 for (size_t i = 0 ; i < firstSequence.getSizeOfSequence();++i){
2     if( isPositionAncestral(i,firstSequence) && isPositionAncestral(i,secondSequence)){
3     return true;
4 }
```

Il est possible de regrouper les tests dans une seule boucle, mais cette manière est la plus efficace. En effet, comme la majorité des paires de séquences ne sont pas compatibles, la première boucle capturera rapidement les séquences incompatibles. Comme il est relativement peu probable que le test échoue parce qu'il n'y a aucun matériel ancestral commun, nous effectuons ce test dans un deuxième temps. Si nous avions réalisé ces tests en même temps, nous n'aurions pas pu éviter l'évaluation de conditions superflues à chaque tour de boucle. Comme la performance est cruciale, nous préférons procéder de cette manière. Nous utilisons des fonctions auxiliaires à la ligne 16 dans le but d'améliorer la lisibilité du code. Voici comment est définie cette fonction :

```
1 inline static bool isPositionAncestral (size_t position, const SnipSequence &
      aSnipSequence){
2 return (!(aSnipSequence.AncestralDataMask[position / BIT_SIZE_OF_DATA_TYPE] &
      BIT_INDICATOR >> (position % BIT_SIZE_OF_DATA_TYPE)));
3 }
```

Cette fonction retourne la valeur vrai si le bit situé au *locus* désiré pour le masque des positions ancestrales est «1». Donc en somme, les fonctions ci-dessus sont équivalentes à l'équation (3.5) qui définit le test pour une coalescence stricte. On

peut voir à la figure 6.3 un léger avantage pour l'algorithme parallèle. Toutefois, il s'agit d'un cas bien spécifique, car les tests sont faits sur des séquences de quinze millions de SNPs uniquement.

6.2.3 Calcul des coalescences

Maintenant que nous avons défini comment on peut établir que deux séquences peuvent coalescer, voici l'implémentation de l'équation 3.6 qui évalue le résultat d'une coalescence :

```
1  inline friend SnipSequence operator+ (const SnipSequence& S1, const SnipSequence& S2) {
2      return  SnipSequence( ((~S2.AncestralDataMask) & S2.snipsData) | ((~S1.
           AncestralDataMask)&S1.snipsData),
3      S1.AncestralDataMask & S2.AncestralDataMask, S1.getSizeOfSequence() );
4  }
```

La structure de données contenant les séquences effectue automatiquement les opérations sur toute la longueur de la séquence, ce qui évite d'écrire une boucle explicitement. Nous avons aussi composé une version parallèle de cette fonction pour quantifier le gain de performance qu'il est possible d'obtenir.

6.2.3.1 Parallélisation des coalescences

Nous avons utilisé du parallélisme de boucle avec les fonctions «*parallel_for*» et «*blocked_range*» de la bibliothèque TBB. Voici un extrait de l'implémentation de cette fonction :

```
1  // Parallel mode.
2  if(SnipSequence::mode== SnipSequence::Mode::PARALLEL){
3      tbb::parallel_for(
4      tbb ::blocked_range<size_t>(0,SnipSequence::getnumberOfBlock() ),
5      [&A,&B,&C,&D,&newMask,&newSnipData]( tbb ::blocked_range<size_t> r ){
6          for( size_t i = r.begin() ; i < r.end() ; ++i){
7              newMask[i] = (A[i])&(C[i]);
8              newSnipData[i] = ((~(C[i]))&(D[i]))|((~(A[i]))&(B[i]));
9          }
10     });
```

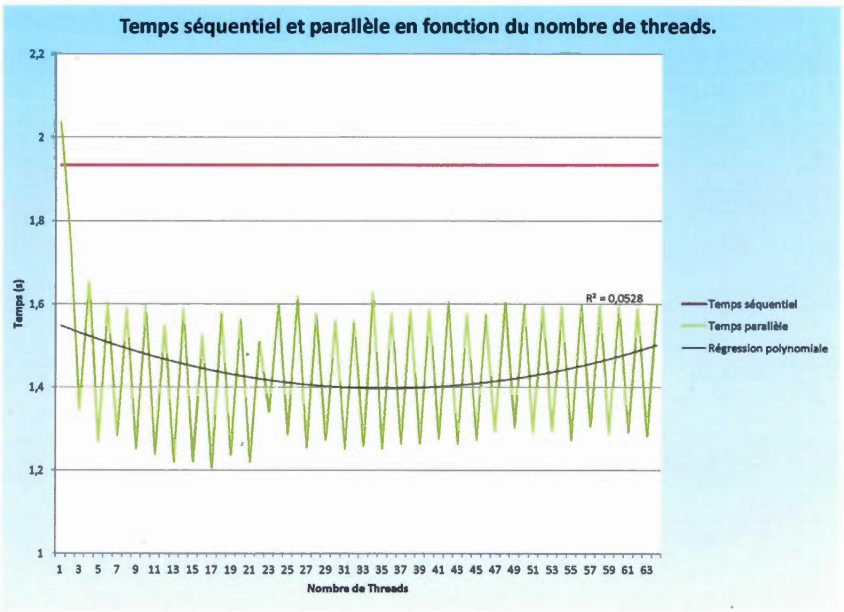


Figure 6.3 Temps séquentiel et parallèle en fonction du nombre de threads exécutés sur le serveur Japet de l'UQAM. Les données ont été obtenues lors du calcul de 50 coalescences de 15 millions de SNPs.

Il s'agit d'une implémentation où la taille des sous-problèmes est gérée dynamiquement. Le masque des *loci* ancestraux et l'état allélique de la séquence produite sont générés sur les lignes 7 et 8. Pour tester l'efficacité de cette fonction, nous avons comparé les temps d'exécution en effectuant cinquante coalescences sur des séquences de diverses longueurs. On commence à observer des gains intéressants lorsque les séquences ont plus de quinze millions de SNPs. La figure 6.3 montre bien le gain de performance que nous obtenons. Toutefois, la fraction du temps consacré au calcul des coalescences est minime par rapport au processus complet de création d'un ARG. Donc, l'amélioration de temps d'exécution pour cette unique opération ne représente pas un gain assez profitable et nous n'avons pas retenu cette approche pour *FastARG*.

On voit à la figure 6.3 que le temps séquentiel est plus lent que les temps parallèles lorsqu'il y a un seul *thread*, car il y a un surcoût associé à la parallélisation du code. Dans tous les tests, nous observons que lorsque le nombre de processus est impair, la performance semble meilleure. Il semble aussi y avoir la présence d'une courbe en «U» mais faible qui dénote le comportement normal d'une fonction parallélisée. Ce qui est bien traduit par la courbe de régression.

6.2.3.2 Coalescences sur carte graphique

Nous avons aussi réalisé une implémentation avec *OpenCl* qui permet d'envoyer du code pour être exécuté par un processeur graphique. Ces unités de traitement possèdent un grand nombre de cœurs (plus de 1000 dans certains cas) pouvant exécuter un grand nombre d'opérations simultanément. En contrepartie, ces cœurs doivent exécuter la même instruction, et ce au même moment. Bien que cela soit une contrainte énorme dans la plupart des contextes, lorsqu'on traite des images où chaque pixel doit subir le même traitement, ce type d'accélération matériel devient incontournable. Comme les fonctions binaires sur les séquences que nous avons développées peuvent être appliquées simultanément sur toute la longueur des séquences, nous avons voulu voir si des gains substantiels étaient possibles en utilisant la carte graphique. Toutefois, écrire du code dans ce paradigme n'est pas trivial et exige un travail colossal même pour une fonction écrite en quelques lignes en programmation séquentielle. En particulier, une simple addition des éléments contenus dans un tableau exige plusieurs centaines de lignes de code [sic], ce qui normalement peut se faire en trois lignes dans un contexte séquentiel. Les gains doivent donc être importants pour que cette avenue soit entreprise. Un programme parallèle pour carte graphique est essentiellement divisé en deux parties. Soit le code maître qui interface le processeur central aux données en entrées et sorties de la carte graphique et le noyau qui lui est exécuté sur les nombreux cœurs présents. Voici le noyau que nous avons envoyé sur le processeur de la carte graphique qui

effectue le calcul du résultat de la coalescence entre deux séquences :

```
1  __kernel void GPU_coal ( __global const unsigned int* A,
2  __global const unsigned int* B,
3  __global const unsigned int* C,
4  __global const unsigned int* D,
5  __global unsigned int* newMask,
6  __global unsigned int* newSnipData,
7  unsigned int N )
8  {
9      int i = get_global_id(0);
10     if (i < N){
11         newMask[i] = (A[i])&(C[i]);
12         newSnipData[i] = ((~(C[i]))&(D[i]))|((~(A[i]))&(B[i]));
13     }
14 }
```

Il est intéressant de noter que le noyau ne possède aucune boucle. L'indexage des opérations est réalisé en partie par la variable située sur la ligne 9. Le code maitre qui est responsable de l'exécution du noyau est présenté à l'annexe D. Il comporte plus de 200 lignes de codes pour la partie entrée et sortie des données seulement. La figure 6.4 illustre les performances que nous obtenons pour cette implantation. Bien que nous avons dû composer un noyau non optimal pour pouvoir le déboguer, les gains ne semblent pas être à la hauteur des attentes. Le calcul d'une coalescence, bien que parallélisable, constitue un problème trop petit sur le plan du traitement pour être évalué de la sorte. Et puisqu'il est extrêmement ardu de compiler des noyaux pour la carte graphique [1], cette avenue n'a pas été retenue pour *FastARG*.

1. http://stackoverflow.com/questions/9464190/error-code-11-what-are-all-possible-reasons-of-getting-error-cl-build-prog

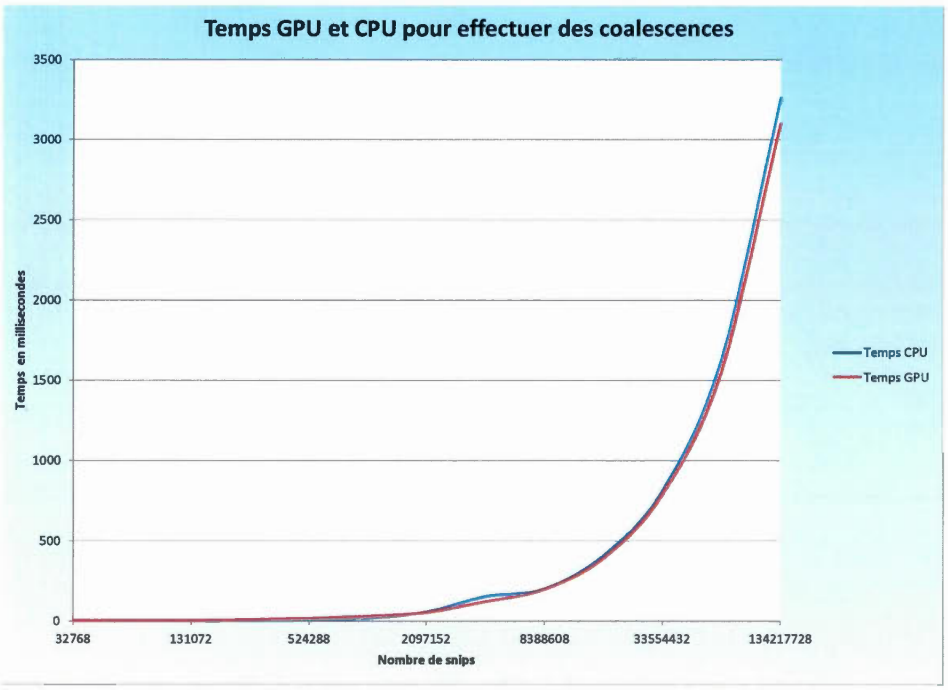


Figure 6.4 Données obtenues avec la fonction BITWISEOPERATIONSONGPU() *qui effectue le calcul de dix coalescences pour plusieurs longueurs de séquences différentes. Les résultats donnent un léger avantage pour le GPU.*

6.2.4 Implémentation des mutations et des recombinaisons

Essentiellement, les mutations changent simplement le bit de l'état allélique du *locus* correspondant de «1» vers «0». Voici la fonction qui joue ce rôle sans les validations normalement effectuées :

```
1 SnipSequence SnipSequence::mutateAPosition( const SnipSequence &
      aSnipSequence, size_t thePosition){
2     size_t blockNumber = thePosition / BIT_SIZE_OF_DATA_TYPE;
3     size_t positionInTheBlock = thePosition % BIT_SIZE_OF_DATA_TYPE;
4     DATA_TYPE bit1OnTheRightPosition= BIT_INDICATOR >> positionInTheBlock;
5     // Creates a new Snip sequence
6     std::valarray<DATA_TYPE> newSnipData = aSnipSequence.snipsData;
7     std::valarray<DATA_TYPE> newMaskData = aSnipSequence.AncestralDataMask;
```

```
8       newSnipData[blockNumber] ^= bit1OnTheRightPosition;// Flips the right
            bit!
9       return SnipSequence( newSnipData , newMaskData, aSnipSequence.
            sizeOfSequence );
10  }
```

Les deux premières variables s'occupent de positionner un bit à l'état «1» sur la bonne position du bloc voulu. Puis on effectue le changement d'état du marqueur mutant à la ligne 8. La création de séquences recombinantes utilise essentiellement les mêmes astuces que pour les dernières opérations. Nous ne présentons donc pas cette fonction. Cependant, nous présentons plutôt dans la section suivante la façon dont un évènement de recombinaison est effectué au niveau de l'ARG.

6.2.4.1 Implémentation d'un évenement de recombinaison

Bien qu'il soit facile de créer les séquences découlant d'une recombinaison, intégrer ces séquences et mettre à jour les variables qui composent notre structure de données ARG est plus complexe. Nous présentons cette procédure uniquement pour un évènement de recombinaison. Bien qu'il existe des différences mineures lorsque d'autres évènements surviennent, la procédure reste essentiellement identique. Nous avons donc morcelé les étapes accomplissant cette tâche et nous les décrivons sommairement. Notons que nous avons retiré la plupart des validations normalement effectuées pour faciliter la compréhension du code. Premièrement, voici l'extrait de la fonction responsable de la création des nouveaux nœuds :

Implémentation d'un évènement de recombinaison

```
1  // We find and iterator on the node that will recombine.
2  vector<size_t>::iterator it;
3  it = find(this->positionOfAliveNodes.begin(), this->positionOfAliveNodes.
       end(), sourceNode );
4
5  // Then we create an array of SnipSequence to receive the result of the
       recombination.
6  try {
7      std::pair<SnipSequence,SnipSequence> RecombinationResultSequences =
           SnipSequence::recombinationOnAposition(this->ARGnodes.at(*it).
           sequence, positionOfTheRecombination);
8      ARGnode leftNode  (ARGNodeType::RECOMBINATIONODE,
           RecombinationResultSequences.first, sourceNode,
           positionOfTheRecombination, {});
9      ARGnode rightNode (ARGNodeType::RECOMBINATIONODE,
           RecombinationResultSequences.second, sourceNode,
           positionOfTheRecombination, {});
10     // Adds the left nodes in the ARG.
11     this->ARGnodes.push_back(leftNode);
12     // Replaces one alive node with the position of the left one.
13     (*it) = this->ARGnodes.size()-1;
14     // Adds the right one in the ARG.
15     this->ARGnodes.push_back(rightNode);
16 }catch(const std::exception & exception){
17     throw std::runtime_error(string(__FILE__) + ':' + std::to_string(
           __LINE__) + "_:_An_exception_has_been_catch_while_creating_the_new_
           recombination_nodes.\n" + exception.what() );
18
19     // Adds the other alive node in the vector.
20     this->positionOfAliveNodes.push_back(this->ARGnodes.size()-1);
21
22 }
```

Comme un évènement de recombinaison ajoute deux séquences à la génération, on ajoute ces nœuds aux lignes 13 et 20. Puis on ajoute les nouveaux nœuds à l'ensemble des nœuds de l'ARG aux lignes 11 et 15.

Implémentation d'un évènement de recombinaison

```
23 // Adds on event on the counter.
24 ++this->totalNumberOfRecombinationEvent;
25
26 // We delete the souce nodes coalescence pairs.
27 for (auto itr = this->possibleCoalescence.begin(); itr != this->
       possibleCoalescence.end(); ){
28 ( (*itr).first == sourceNode || (*itr).second == sourceNode ) ? ( itr =
       this->possibleCoalescence.erase(itr) ) : (++itr);
29 }
```

Puis on ajoute au compteur d'évènements une recombinaison. Notre structure de données contient une matrice contenant toutes les possibilités de coalescence. Donc, les lignes 27 à 29 se chargent de retirer toutes les coalescences avec la séquence d'origine.

Implémentation d'un évènement de recombinaison

```
30 // We then add all the coalescence pairs for the new node.
31 for ( size_t i = 0 ; i < this->positionOfAliveNodes.size() ; ++i){
32     if (this->positionOfAliveNodes.at(i) != this->ARGnodes.size()-1  ){
33         if (this->ARGnodes.back().sequence == this->ARGnodes.at(this->
               positionOfAliveNodes.at(i)).sequence){
34             this->possibleCoalescence.insert(std::pair<size_t,size_t>{this
                   ->positionOfAliveNodes.at(i), this->ARGnodes.size()-1});
35         }
36     }
37     if (this->positionOfAliveNodes.at(i) != this->ARGnodes.size()-2){
38         if (this->ARGnodes.at(this->ARGnodes.size()-2).sequence == this->
               ARGnodes.at(this->positionOfAliveNodes.at(i)).sequence){
39             this->possibleCoalescence.insert(std::pair<size_t,size_t>{this
                   ->positionOfAliveNodes.at(i), this->ARGnodes.size()-2});
40         }
41     }
42 }
43 this->numberOfCoalescencePossible = this->possibleCoalescence.size();
```

Puis on ajoute toutes les coalescences possibles avec les deux nouvelles séquences générées. Les deux conditions présentes aux lignes 32 et 37 testent si chacune des deux nouvelles séquences peuvent coalescer avec les autres séquences présentes dans la génération courante. Une seule variable d'importance n'est pas mise à jour

lors d'une recombinaison. Il s'agit du vecteur contenant le nombre de marqueurs mutants à chaque *locus*. Ce dernier n'est pas affecté lors d'une recombinaison, mais il doit être mis à jour lors d'une mutation ou une coalescence. Voici donc un extrait de la fonction générant un évènement de coalescence qui est responsable de la mise à jour de ce vecteur :

```
1
2  // Populates the vector that contains the number of mutant locus for each
       position.
3  for (size_t i = 0 ; i < this->lenghtOfSequences; ++i){
4      // we decrement by 1 if BOTH locus were mutant.
5      if ( SnipSequence::isPositionMutant( i ,this->ARGnodes.at(
           firstSourceNode).sequence ) &&
6           SnipSequence::isPositionMutant( i ,this->ARGnodes.at(
               secondSourceNode).sequence ) ){
7           --this->numberOfMutantSNPsByPosition.at(i);
8           // We check if we can mutate this locus after decrementing.
9           if ( this->numberOfMutantSNPsByPosition.at(i) == 1 ){
10              ++this->numberOfMutationPossible;
11              this->isAMutationEventPossible = true;
12          }
13     }
14 }
```

Bien que cette section représente une petite partie de l'implémentation, nous croyons qu'elle est grandement représentative de l'ensemble.

6.2.5 Extraction des topologie

Un des grands avantages de *FastARG* est la possibilité d'extraire toutes les topologies possibles pour chacun des nœuds, et ce pour tous les *loci*. On entend par topologie l'ensemble des personnes ayant hérité d'un SNPs à travers les générations. Clairement, les évènements de mutation n'affectent pas les topologies et elles restent donc identiques pour le nœud parent. Les coalescences fusionnent deux nœuds, donc les topologies sont elles aussi fusionnées. S'il s'agit d'une sé-

quence ayant recombiné, la topologie dépendra uniquement de la position pour laquelle on souhaite extraire la topologie. Si le marqueur est ancestral, les nœuds parents ont les mêmes topologies que leur enfant. Dans le cas contraire, la topologie est inexistante, car nous n'avons pas d'information sur ce marqueur. Nous avons donc défini une fonction récursive qui procède selon le type de nœud :

```
1  vector<uint16_t> ARG::getLeafReachabilityforALocus(size_t theLocusPosition,
       size_t nodeIndex ) const {
2      if (this->ARGnodes.at(nodeIndex).typeOfNode == ARGNodeType::
           LEAFSAMPLEMODE ||
3          this->ARGnodes.at(nodeIndex).typeOfNode == ARGNodeType::
               COALESCENSENODE ){
4          return this->ARGnodes.at(nodeIndex).leafReachabilityForALocus.at(
               theLocusPosition);
5      }else if(this->ARGnodes.at(nodeIndex).typeOfNode == ARGNodeType::
           MUTATIONODE){
6          return getLeafReachabilityforALocus( theLocusPosition, this->
               ARGnodes.at(nodeIndex).positionOfDataSource);
7      }else{ // We are in a recombination node.
8          if ( SnipSequence::isPositionAncestral(theLocusPosition, this->
               ARGnodes.at(nodeIndex).sequence)){
9              return getLeafReachabilityforALocus(theLocusPosition, this->
                   ARGnodes.at(nodeIndex).positionOfDataSource);
10         }else{
11             return {}; // C++11
12         }
13     }
14 }
```

Les feuilles possèdent une topologie singleton composée d'un seul nœud racine dont elles sont la source. Puis lorsqu'un événement de coalescence survient, on fait l'union des topologies de la manière suivante :

```
1   vector<uint16_t> ARG::getLea
2  // In this case we need to make the unions of the two nodes.
3  vector<vector<uint16_t>> resultVectorForLeafReach;
4  vector<uint16_t> resultVectorForLeafReachForALocus;
5  vector<uint16_t> leafReachForFirstNodeForALocus,
       leafReachForSecondNodeForALocus;
6
7  for (size_t i = 0 ; i < this->lenghtOfSequences  ; ++i){
8      leafReachForFirstNodeForALocus = getLeafReachabilityforALocus(i,
           firstSourceNode);
9      leafReachForSecondNodeForALocus = getLeafReachabilityforALocus(i,
           secondSourceNode);
10
11     std::set_union(leafReachForFirstNodeForALocus.begin(),
           leafReachForFirstNodeForALocus.end(),
12     leafReachForSecondNodeForALocus.begin(),
           leafReachForSecondNodeForALocus.end(),
13     std::back_inserter(resultVectorForLeafReachForALocus));
14     resultVectorForLeafReach.push_back(resultVectorForLeafReachForALocus);
15     resultVectorForLeafReachForALocus.clear();
16 }
```

Nous avons choisi de procéder de cette façon, car cela minimise la quantité de calcul nécessaire. *FastARG* a donc accès rapidement à l'ensemble des topologies pour tous les nœuds des graphes qu'il produit.

6.2.6 Implémentation d'algorithme générale

Avec toutes ces fonctions en place, il devient aisé de définir un algorithme en spécialisant les fonctions et les classes de *FastARG*. Par exemple, nous avons vu à la section 5.3.1 que l'heuristique de Margarita sélectionne les séquences à recombiner selon certains critères précis. Il suffit alors d'implémenter ces fonctions en héritant de notre classe ARG pour créer des ARGs de type Margarita. Nous avons implémenté toutes les fonctions nécessaires à cela en trois cents lignes de code environ. Cela est très peu si on compare à une implémentation de *novo*. De plus, comme tous les algorithmes utiliseront par défaut l'encodage et les méthodes de

traitement binaires des séquences, *FastARG* possède une assise solide, car toutes ses opérations constitutives ont été optimisées et vérifiées avec attention.

Puisque le prototypage et la mise en œuvre d'algorithmes sont grandement simplifiés par *FastARG*, on peut facilement explorer de nouvelle piste pour la création d'heuristiques de constructions d'ARGs. On peut voir à l'annexe B l'implémentation en *C++* que nous avons faite de l'algorithme de Margarita à la section 5.3.1 que nous avons développé rapidement grâce à *FastARG*. Le même processus a permis de développer assez rapidement l'algorithme Manhattan.

6.3 Les statistiques

Maintenant que les ARGs sont bien définis, nous présentons dans cette section quelques stratégies pour tirer profit de l'information contenue dans les ARGs. De plus, nous mettons l'accent sur l'apport de *FastARG* facilitant le calcul de certaines statistiques.

6.3.1 Test du χ^2

Pour tester l'association entre un *locus* et un phénotype avec le graphe de recombinaison ancestral, il suffit de poser un TIM près d'un *locus* pour un nœud choisi et d'observer quelles sont les séquences génétiques de l'échantillon qui l'obtiendraient. Comme on connait le phénotype des séquences qui correspond aux feuilles du graphe, on peut construire une table de contingence.

Le test du χ^2 se fait uniquement à l'aide des données en entrée. Donc il ne tire pas avantage de l'information supplémentaire générée par l'ARG. Dans ce cas, il

est toujours possible de faire un simple test du χ^2 entre le phénotype et l'état mutant ou primitif d'un allèle. Voici un exemple d'une statistique simple fait sur les séquences contenues dans les feuilles seulement. Il suffit d'observer la correspondance entre un marqueur et le phénotype d'un individu. Le tableau 6.2 illustre le test du χ^2 pour le marqueur 3 du graphe de la figure 2.3.

Tableaux 6.2 Test du χ^2 pour le marqueur 3 du graphe de la figure 2.3

	Cas	Témoins	
Mutant	4	2	6
Primitif	0	2	2
	4	4	8

(a) La table des observations.

	Cas	Témoins	
Mutant	3	3	6
Primitif	1	1	2
	4	4	8

(b) La table des estimations.

Pour trouver la position la plus probable, il suffit de prendre celle dont la valeur-p est la plus petite. Voici le résultat pour le tableau 6.2 :

$$\chi^2 = \sum_{i=1}^{r}\sum_{j=1}^{c} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}} \text{ où } E_{i,j} = Np_ip_j,$$

$$\chi^2 = \frac{(4-3)^2}{3} + \frac{(2-3)^2}{3} + \frac{(0-1)^2}{1} + \frac{(2-1)^2}{1} = \frac{1}{3} + \frac{1}{3} + 2 = 2.\bar{6},$$

$$\chi^2 = 2.\bar{6}, \qquad \text{valeur-p} = 0.10247.$$

6.3.2 Test par percolation

Un test un peu plus élaboré utilisé par Margarita consiste à faire percoler les mutations à partir des nœuds internes d'un ARG jusqu'aux feuilles. Comme on connaît le phénotype des individus, on peut retester à l'aide du χ^2 la plausibilité de cette mutation. On peut alors prendre une statistique qui déterminera

la position la plus probable pour un TIM. Par exemple, on peut opter pour la moyenne des tests à chaque *locus* testé et choisir la position où on obtient la meilleure moyenne. On pourrait au lieu de la moyenne prendre uniquement la valeur la plus élevée. Cette méthode permet d'utiliser l'information générée par la construction de l'ARG. Toutefois, comme une quantité de calculs considérable est nécessaire si les topologies n'ont pas été générées a priori, cette étape peut devenir contraignante. *FastARG* calcule toutes ces topologies en supposant que le point de recombinaison a été effectué au milieu des intervalles entre chacun des marqueurs. La figure 6.5 illustre bien comment les mutations se propagent à travers un ARG. On peut aussi voir le rôle que les recombinaisons jouent pour la structuration des topologies possibles. Par exemple, on peut voir que la mutation sur le nœud 25 ne se propage pas vers le nœud 24, mais seulement vers le nœud 17. Ceci est dû au fait que la mutation est située dans une zone non ancestrale du nœud 24, mais dans une zone ancestrale pour le nœud 17.

6.3.3 Constructions de nouveaux graphes

Une autre méthode possible consiste à ajouter aux données initiales un marqueur à toutes les séquences de l'échantillon étant affecté et un marqueur primitif aux séquences contrôle. Puis en construisant des ARGs avec ces marqueurs annexés, on teste la vraisemblance des graphes générés. Comme un ARG devrait être plus cohérent si le marqueur est bien positionné, cela devrait se refléter dans le test de vraisemblance. Bien que cette méthode semble être la plus naturelle, elle requiert une grande quantité de calculs pour chaque test. Donc le temps de calcul devient vite prohibitif même pour des échantillons de taille modeste. Mais contrairement à la méthode précédente où les mutations inférées ne changent pas la topologie de l'ARG, cette méthode permet de valider plus adéquatement la position d'un TIM. La vraisemblance testant la plausibilité de ce type d'ARG peut aussi faire intervenir le temps nécessaire pour parvenir au MRCA. En effet, lors de la construction de

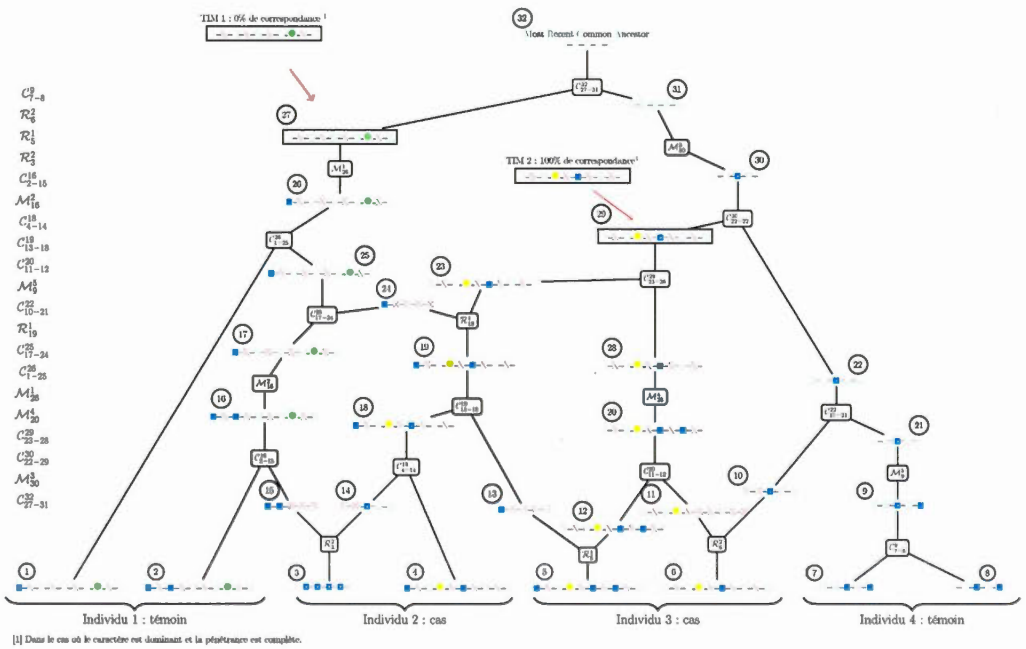


Figure 6.5 Illustration de la propagation de deux mutations vers les séquences observées. À gauche, on retrouve les opérations effectuées pour construire cet ARG. Les «X» en rouge désignent les points de recombinaison.

chaque graphe il est possible d'ajouter une mesure de temps sur chaque branche, ce qui permet d'avoir une estimation du temps avant de parvenir à un MRCA. Puis, en les comparant par exemple aux résultats théoriques comme dans l'équation (4.12), on peut dériver une vraisemblance. Ceci correspond globalement à la technique employée par Fearnhead et Donnely (2001).

6.4 Résultats

Nous avons contacté les auteurs de l'heuristique ARG4WG pour obtenir les jeux de données que les auteurs ont utilisés dans leur article afin de comparer la performance de leur implémentation. Nous souhaitions aussi comparer le nombre de recombinaisons produites, car cela est une métrique importante de la qualité des graphes. Voici donc trois figures représentant les temps d'exécution pour trois des algorithmes que nous avons implémentés ainsi que le nombre de recombinaisons nécessaires à chacun des tests. Les tableaux 6.3 contient l'ensemble des tests réalisés avec 500 haplotypes. Les tableaux 6.4 et 6.5 quant à eux contiennent les résultats obtenus pour 1000 et 2000 haplotypes respectivement. Notons que les temps d'exécution que nous obtenons versus les temps d'exécution que les auteurs obtiennent pour leur implémentation sont difficilement comparables. Premièrement, le système sur lequel ils ont été exécutés n'est pas dans la même catégorie. Mais surtout leur implémentation ne calcule pas les mêmes métadonnées sur chacun des ARGs produits. Les résultats présentés permettent de valider le bon fonctionnement de *FastARG*.

On peut constater assez clairement que l'algorithme de Margarita semble produire beaucoup plus de recombinaisons et est donc beaucoup plus lent. Les temps d'exécution de Margarita ne sont toutefois pas proportionnels. Cela est probablement dû au fait que le calcul effectué pour choisir les recombinaisons exige beaucoup

Tableaux 6.3 Temps d'exécution et nombre de recombinaisons pour des échantillons contenant 500 *haplotypes pour* 1000, 2000 *et* 5000 *SNPs.*

	1000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	325	4865	139	2648	1	3383
ARG4WG	10	3761	5	1958	7	2478
Manhattan	6	3740	3	1965	2	2454

(a) 1000 SNPs par 500 haplotypes.

	2000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	1368	10108	941	7193	955	7144
ARG4WG	39	7543	27	5474	26	5236
Manhattan	20	7594	14	5506	13	5276

(b) 2000 SNPs par 500 haplotypes.

	5000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	10077	24062	6076	17885	6872	18132
ARG4WG	241	17843	180	12769	183	13226
Manhattan	148	18135	114	1309	115	13533

(c) 5000 SNPs par 500 haplotypes.

plus de travail que dans le cas d'ARG4WG et Manhattan.

On peut constater que le passage de 500 haplotypes vers 1000 haplotypes est moins pénalisant pour Manhattan que pour ARG4WG. Manhattan semble plutôt linéaire en complexité temporelle tandis qu'on remarque un ralentissement de

Tableaux 6.4 Temps d'exécution et nombre de recombinaisons pour des échantillons contenant 1000 *haplotypes pour* 1000, 2000 *et* 5000 *SNPs.*

	1000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	1622	8401	694	4792	967	5631
ARG4WG	30	6403	15	3484	19	4160
Manhattan	14	6367	10	3451	8	4055

(a) 1000 SNPs par 1000 haplotypes.

	2000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	6989	17696	4830	12706	4526	12044
ARG4WG	106	13086	74	9558	69	8887
Manhattan	48	13180	34	9586	30	8849

(b) 2000 SNPs par 1000 haplotypes.

	5000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	48869	42242	30417	31138	33453	31199
ARG4WG	741	31344	415	21706	441	22945
Manhattan	391	31377	188	21999	202	2363

(c) 5000 SNPs par 1000 haplotypes.

l'algorithme ARG4WG.

Tableaux 6.5 Temps d'exécution et nombre de recombinaisons pour des échantillons contenant 2000 *haplotypes pour* 1000, 2000 *et* 5000 *SNPs.*

	1000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	11054	14951	4621	8553	6235	10062
ARG4WG	100	11384	55	6220	67	7372
Manhattan	35	11108	22	6080	23	7190

(a) 1000 SNPs par 2000 haplotypes.

	2000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	47476	31080	32469	22567	29508	21521
ARG4WG	334	23169	253	16760	231	15706
Manhattan	110	22888	88	16661	79	15613

(b) 2000 SNPs par 2000 haplotypes.

	5000 SNPs					
	Échantillon 1		Échantillon 2		Échantillon 3	
	Temps (s)	(Re)	Temps (s)	(Re)	Temps (s)	(Re)
Margarita	N/A	N/A	N/A	N/A	N/A	N/A
ARG4WG	N/A	N/A	1327	37252	1466	40869
Manhattan	N/A	N/A	562	37542	625	41231

(c) 5000 SNPs par 2000 haplotypes.

Les tendances observées précédemment se maintiennent, cependant les échantillons contenant 5000 SNPs ont surpassé les limites en mémoire du système sur lequel ces tests ont été effectués. Ces résultats semblent plutôt positifs pour Manhattan et *FastARG* et ils démontrent l'efficience de notre implémentation.

CONCLUSION

Le problème que nous avons exploré lors de notre recherche, soit la génération efficiente de graphes de recombinaison ancestraux, se situe au carrefour de trois domaines principaux. Premièrement la biologie, où la problématique prend racine. Puis, plusieurs branches des mathématiques telles que la statistique et l'algèbre nous fournissent la plupart des outils essentiels pour modéliser et analyser la quantité sans cesse croissante de données biologiques dont nous disposons. Enfin, comme la qualité des résultats obtenus à l'aide des graphes est directement proportionnelle à la quantité de données que les algorithmes sont aptes à traiter, une implémentation informatique de qualité est de mise. Plusieurs aspects de l'informatique tant théoriques que pratiques permettent d'optimiser le traitement des données pour que nous soyons en mesure d'obtenir des temps d'exécution raisonnables. L'implémentation de *FastARG* [1] est donc le fruit de multiples résultats provenant de ces trois domaines.

Bien que nos tentatives de parallélisation n'aient pas été fructueuses, les opérations séquentielles semblent avoir des performances prometteuses. Toutefois, l'ensemble des métadonnées produites par *FastARG* représente un avantage majeur pour le calcul de statistiques découlant de ces graphes. Nous sommes donc persuadés que *FastARG* ouvre une nouvelle approche computationnelle pour la génération de graphes de recombinaison ancestraux comportant un grand nombre de marqueurs

1. `https://bitbucket.org/fastarguqam/` - (13 septembre 2018)

génétiques.

Ce travail peut donc avoir des implications intéressantes en cartographie génétique, car les ARGs sont un des outils principaux permettant d'inférer la position des gènes.

ANNEXE A

DÉMONSTRATIONS SUPPLÉMENTAIRES DE LA FONCTION TESTANT L'INCOMPATIBILITÉ D'UN LOCUS.

Démonstration pour le locus 1 :

$$\begin{aligned} f(\mathcal{S}[1], \mathcal{S}'[1]) &= (\neg\mathcal{A}[1]) \wedge (\neg\mathcal{A}'[1]) \wedge (((\neg\mathcal{M}[1]) \wedge \mathcal{M}'[1]) \vee (\mathcal{M}[1] \wedge (\neg\mathcal{M}'[1])))\\ &= (\neg 0) \wedge (\neg 0) \wedge (((\neg 0) \wedge 0) \vee (0 \wedge (\neg 0)))\\ &= 1 \wedge 1 \wedge ((1 \wedge 0) \vee (0 \wedge (1)))\\ &= 1 \wedge 1 \wedge (0 \vee 0))\\ &= 1 \wedge 1 \wedge 0 = 0. \end{aligned}$$

Démonstration pour le locus 9 :

$$\begin{aligned} f(\mathcal{S}[9], \mathcal{S}'[9]) &= (\neg\mathcal{A}[9]) \wedge (\neg\mathcal{A}'[9]) \wedge (((\neg\mathcal{M}[9]) \wedge \mathcal{M}'[9]) \vee (\mathcal{M}[9] \wedge (\neg\mathcal{M}'[9])))\\ &= (\neg 0) \wedge (\neg 0) \wedge (((\neg 1) \wedge 0) \vee (1 \wedge (\neg 0)))\\ &= 1 \wedge 1 \wedge ((0 \wedge 0) \vee (1 \wedge (1)))\\ &= 1 \wedge 1 \wedge (0 \vee 1))\\ &= 1 \wedge 1 \wedge 1 = 1. \end{aligned}$$

Démonstration pour le locus 16 :

$$\begin{aligned} f(\mathcal{S}[16], \mathcal{S}'[16]) &= (\neg\mathcal{A}[16]) \wedge (\neg\mathcal{A}'[16]) \wedge (((\neg\mathcal{M}[16]) \wedge \mathcal{M}'[16]) \vee (\mathcal{M}[16] \wedge (\neg\mathcal{M}'[16]))) \\ &= (\neg 1) \wedge (\neg 1) \wedge (((\neg 1) \wedge 1) \vee |(1 \wedge (\neg 1))) \\ &= 0 \wedge 0 \wedge ((0 \wedge 1) \vee (1 \wedge (0))) \\ &= 0. \end{aligned}$$

ANNEXE B

IMPLÉMENTATION DE L'ALGORITHME DE MARGARITA

Manhattan.cpp

```
1  void Manhattan::run(void){
2      boost::progress_display show_progress( numberOfPositionToBeMuted );
3      while (this->positionOfAliveNodes.size() > 1 ){ // i.e. While not OMRCA
4          show_progress += this->totalNumberOfMutationEvent - show_progress.count();
5          // If an coalescence and a mutation is possible.
6          if (this->numberOfCoalescencePossible != 0 && this->numberOfMutationPossible
               !=0 ){
7              // We choose proportionally between possible events.
8              std::uniform_int_distribution<size_t>randomLongestSharedEnd(1, this->
                   numberOfCoalescencePossible + this->numberOfMutationPossible);
9              size_t winningTransition = randomLongestSharedEnd(this->eng);
10             // Then we make the "winning" event.
11             if (winningTransition <= this->numberOfCoalescencePossible ){
12                 generateARandomCoalescentEvent();
13             }else{
14                 generateARandomMutationEvent();
15             }
16         }else if (this->numberOfCoalescencePossible == 0 && this->
               numberOfMutationPossible !=0 ){
17             // Case if only a mutation is possible.
18             generateARandomMutationEvent();
19         }else if (this->numberOfCoalescencePossible != 0 && this->
               numberOfMutationPossible ==0 ){
20             // Case if only a coalescence is possible.
21             generateARandomCoalescentEvent();
22         }else{ // Case when we need to recombine.
23             generatesAnManhattanRecombinationEvents();
24         }
25     }
26 }
```

ANNEXE C

IMPLÉMENTATION DES TESTS UNITAIRES

FastARGBoostTests.cpp

```
1  BOOST_AUTO_TEST_CASE (FullArgData)
2  {
3      Parameters params =  Parameters();
4      params.algorithmChoice = algorithms::ARG4WG;
5      params.coalescenceModeChoice = coalescenceMode::HAVE_ANCESTRAL_MATERIAL_IN_COMMON;
6      params.dataFile = "";
7      params.numberOfSequences = 8;
8      params.sequencesSize = 5;
9      vector<size_t> distances {1,1,1,1,1};
10
11     SnipSequence Leaf0("10000");
12     ARGnode Node0(ARGNodeType::LEAFSAMPLEMODE, Leaf0, 0, std::numeric_limits<size_t>::
           max(), {{0},{0},{0},{0},{0}});
13     SnipSequence Leaf1("11000");
14     ARGnode Node1(ARGNodeType::LEAFSAMPLEMODE, Leaf1, 1, std::numeric_limits<size_t>::
           max(), {{1},{1},{1},{1},{1}});
15     SnipSequence Leaf2("11100");
16     ARGnode Node2(ARGNodeType::LEAFSAMPLEMODE, Leaf2, 2, std::numeric_limits<size_t>::
           max(), {{2},{2},{2},{2},{2}});
17     SnipSequence Leaf3("10100");
18     ARGnode Node3(ARGNodeType::LEAFSAMPLEMODE, Leaf3, 3, std::numeric_limits<size_t>::
           max(),{{3},{3},{3},{3},{3}});
19     SnipSequence Leaf4("10110");
20     ARGnode Node4(ARGNodeType::LEAFSAMPLEMODE, Leaf4, 4, std::numeric_limits<size_t>::
           max(),{{4},{4},{4},{4},{4}});
21     SnipSequence Leaf5("00100");
22     ARGnode Node5(ARGNodeType::LEAFSAMPLEMODE, Leaf5, 5, std::numeric_limits<size_t>::
           max(),{{5},{5},{5},{5},{5}});
23     SnipSequence Leaf6("00101");
24     ARGnode Node6(ARGNodeType::LEAFSAMPLEMODE, Leaf6, 6, std::numeric_limits<size_t>::
           max(),{{6},{6},{6},{6},{6}});
25     SnipSequence Leaf7("00101");
26     ARGnode Node7(ARGNodeType::LEAFSAMPLEMODE, Leaf7, 7, std::numeric_limits<size_t>::
           max(),{{7},{7},{7},{7},{7}});
27     vector<bool> phenotypes {0,1,1,0};
28     vector<ARGnode> leaves {Node0, Node1, Node2, Node3, Node4, Node5, Node6, Node7};
29     ARG4WG aArg (params, leaves, phenotypes, distances);
30
31     BOOST_CHECK(aArg.lenghtOfSequences == 5);
32     BOOST_CHECK(aArg.numberOfSequences == 8);
33
34     BOOST_CHECK(aArg.totalNumberOfMutationEvent ==  0);
```

```
35      BOOST_CHECK(aArg.totalNumberOfCoalescenceEvent == 0);
36      BOOST_CHECK(aArg.totalNumberOfRecombinationEvent == 0);
37
38      BOOST_CHECK(aArg.numberOfMutationPossible == 1);
39      BOOST_CHECK(aArg.numberOfPositionToBeMuted == 5);
40
41      BOOST_CHECK(aArg.isAMutationEventPossible);
42      vector<size_t> numberOfMutantLocus = {5,2,6,1,2};
43      BOOST_CHECK(aArg.numberOfMutantSNPsByPosition == numberOfMutantLocus );
44
45      for( uint16_t i = 0 ; i < 8 ; ++i){
46          for( uint16_t j = 0 ; j < 5 ; ++j){
47              vector<uint16_t> leafReachTemp = {i};
48              BOOST_CHECK(aArg.getLeafReachabilityforALocus(j, i) == leafReachTemp );
49          }
50      }
51
52      BOOST_CHECK(aArg.numberOfCoalescencePossible == 1);
53      vector<vector<bool>> coal = {
54          {0,0,0,0,0,0,0,0},
55          {0,0,0,0,0,0,0,0},
56          {0,0,0,0,0,0,0,0},
57          {0,0,0,0,0,0,0,0},
58          {0,0,0,0,0,0,0,0},
59          {0,0,0,0,0,0,0,0},
60          {0,0,0,0,0,0,0,1},
61          {0,0,0,0,0,0,1,0}};
62      // BOOST_CHECK(aArg.coalescencePosibilitiesForLivesNodes == coal);
63
64      vector<size_t> aliveNodes = {0,1,2,3,4,5,6,7};
65      BOOST_CHECK(aArg.positionOfAliveNodes == aliveNodes );
66
67      vector<vector<std::pair<size_t,size_t>>> leftRight = {
68          {{5,5}, {1,1}, {1,2}, {2,2}, {2,3}, {5,2}, {5,5}, {5,5}},
69          {{1,1}, {5,5}, {2,2}, {1,2}, {1,3}, {5,2}, {5,5}, {5,5}},
70          {{1,2}, {2,2}, {5,5}, {1,1}, {1,3}, {5,1}, {5,5}, {5,5}},
71          {{2,2}, {1,2}, {1,1}, {5,5}, {3,3}, {5,0}, {5,5}, {5,5}},
72          {{2,3}, {1,3}, {1,3}, {3,3}, {5,5}, {5,3}, {5,5}, {5,5}},
73          {{5,2}, {5,2}, {5,1}, {5,0}, {5,3}, {5,5}, {4,5}, {4,5}},
74          {{5,5}, {5,5}, {5,5}, {5,5}, {5,5}, {4,5}, {5,5}, {5,5}},
75          {{5,5}, {5,5}, {5,5}, {5,5}, {5,5}, {4,5}, {5,5}, {5,5}}
76      };
77
78      BOOST_CHECK( leftRight == aArg.leftRightSharedIndexMatrixForLiveNodes);
79
80      /*************************************************************************/
81      aArg.generateACoalescentEvent(6,7);
82
83      BOOST_CHECK(aArg.totalNumberOfMutationEvent == 0);
84      BOOST_CHECK(aArg.totalNumberOfCoalescenceEvent == 1);
85      BOOST_CHECK(aArg.totalNumberOfRecombinationEvent == 0);
86
87      BOOST_CHECK(aArg.numberOfMutationPossible == 2);
88      BOOST_CHECK(aArg.numberOfPositionToBeMuted == 5);
89
90      BOOST_CHECK(aArg.isAMutationEventPossible);
91      numberOfMutantLocus = {5,2,5,1,1};
92      BOOST_CHECK(aArg.numberOfMutantSNPsByPosition == numberOfMutantLocus );
93
94      vector<uint16_t> leafReachTemp = {6,7};
95      for( uint16_t j = 0 ; j < 5 ; ++j){
96          BOOST_CHECK(aArg.getLeafReachabilityforALocus(j,8) == leafReachTemp );
97      }
```

```
98
99      BOOST_CHECK(aArg.numberOfCoalescencePossible ==  0);
100     coal = {{0,0,0,0,0,0,0},
101         {0,0,0,0,0,0,0},
102         {0,0,0,0,0,0,0},
103         {0,0,0,0,0,0,0},
104         {0,0,0,0,0,0,0},
105         {0,0,0,0,0,0,0},
106         {0,0,0,0,0,0,0}};
107     // BOOST_CHECK(aArg.coalescencePosibilitiesForLivesNodes == coal);
108
109     aliveNodes = {0,1,2,3,4,5,8};
110     BOOST_CHECK(aArg.positionOfAliveNodes == aliveNodes );
111
112     leftRight = {
113         {{5,5}, {1,1}, {1,2}, {2,2}, {2,3}, {5,2}, {5,5}},
114         {{1,1}, {5,5}, {2,2}, {1,2}, {1,3}, {5,2}, {5,5}},
115         {{1,2}, {2,2}, {5,5}, {1,1}, {1,3}, {5,1}, {5,5}},
116         {{2,2}, {1,2}, {1,1}, {5,5}, {3,3}, {5,0}, {5,5}},
117         {{2,3}, {1,3}, {1,3}, {3,3}, {5,5}, {5,3}, {5,5}},
118         {{5,2}, {5,2}, {5,1}, {5,0}, {5,3}, {5,5}, {4,5}},
119         {{5,5}, {5,5}, {5,5}, {5,5}, {5,5}, {4,5}, {5,5}}
120     };
121
122     BOOST_CHECK( leftRight == aArg.leftRightSharedIndexMatrixForLiveNodes);
123
124     /*************************************************************************/
125     aArg.generateARecombinationEvent(2,5);
126
127     BOOST_CHECK(aArg.totalNumberOfMutationEvent ==  0);
128     BOOST_CHECK(aArg.totalNumberOfCoalescenceEvent ==  1);
129     BOOST_CHECK(aArg.totalNumberOfRecombinationEvent ==  1);
130
131     BOOST_CHECK(aArg.numberOfMutationPossible ==  2);
132     BOOST_CHECK(aArg.numberOfPositionToBeMuted ==  5);
133     BOOST_CHECK(aArg.isAMutationEventPossible);
134
135     numberOfMutantLocus = {5,2,5,1,1};
136     BOOST_CHECK(aArg.numberOfMutantSNPsByPosition == numberOfMutantLocus );
137
138     leafReachTemp = {};
139     BOOST_CHECK(aArg.getLeafReachabilityforALocus(0, 9) == leafReachTemp );
140     BOOST_CHECK(aArg.getLeafReachabilityforALocus(0, 9) == leafReachTemp );
141     BOOST_CHECK(aArg.getLeafReachabilityforALocus(2, 10) == leafReachTemp );
142     BOOST_CHECK(aArg.getLeafReachabilityforALocus(3, 10) == leafReachTemp );
143     BOOST_CHECK(aArg.getLeafReachabilityforALocus(4, 10) == leafReachTemp );
144     leafReachTemp = {5};
145     BOOST_CHECK(aArg.getLeafReachabilityforALocus(0, 10) == leafReachTemp );
146     BOOST_CHECK(aArg.getLeafReachabilityforALocus(1, 10) == leafReachTemp );
147     BOOST_CHECK(aArg.getLeafReachabilityforALocus(2, 9) == leafReachTemp );
148     BOOST_CHECK(aArg.getLeafReachabilityforALocus(3, 9) == leafReachTemp );
149     BOOST_CHECK(aArg.getLeafReachabilityforALocus(4, 9) == leafReachTemp );
150
151     BOOST_CHECK(aArg.numberOfCoalescencePossible ==  3);
152     coal = {{0,0,0,0,0,0,0,0},
153         {0,0,0,0,0,0,0,0},
154         {0,0,0,0,0,1,0,0},
155         {0,0,0,0,0,1,0,0},
156         {0,0,0,0,0,0,0,0},
157         {0,0,1,1,0,0,0,0},
158         {0,0,0,0,0,0,0,1},
159         {0,0,0,0,0,0,1,0}};
160     // BOOST_CHECK(aArg.coalescencePosibilitiesForLivesNodes == coal);
```

```
161
162     aliveNodes = {0,1,2,3,4,9,8,10};
163     BOOST_CHECK(aArg.positionOfAliveNodes == aliveNodes );
164
165     leftRight = {
166         {{5,5}, {1,1}, {1,2}, {2,2}, {2,3}, {5,2}, {5,5}, {5,0} },
167         {{1,1}, {5,5}, {2,2}, {1,2}, {1,3}, {5,2}, {5,5}, {5,5} },
168         {{1,2}, {2,2}, {5,5}, {1,1}, {1,3}, {5,5}, {5,5}, {5,5} },
169         {{2,2}, {1,2}, {1,1}, {5,5}, {3,3}, {5,5}, {5,5}, {5,0} },
170         {{2,3}, {1,3}, {1,3}, {3,3}, {5,5}, {3,3}, {5,5}, {5,0} },
171         {{5,2}, {5,2}, {5,5}, {5,5}, {3,3}, {5,5}, {4,5}, {5,5} },
172         {{5,5}, {5,5}, {5,5}, {5,5}, {5,5}, {4,5}, {5,5}, {5,5} },
173         {{5,0}, {5,5}, {5,5}, {5,0}, {5,0}, {5,5}, {5,5}, {5,5} }
174     };
175
176     BOOST_CHECK( leftRight == aArg.leftRightSharedIndexMatrixForLiveNodes);
177     /*************************************************************************/
178
179
180     ... //
181
182
183
184     /*************************************************************************/
185     aArg.generateACoalescentEvent(26,30);
186
187     BOOST_CHECK(aArg.totalNumberOfMutationEvent ==  5);
188     BOOST_CHECK(aArg.totalNumberOfCoalescenceEvent ==  11);
189     BOOST_CHECK(aArg.totalNumberOfRecombinationEvent ==  4);
190
191     BOOST_CHECK(aArg.numberOfMutationPossible ==  0);
192     BOOST_CHECK(aArg.numberOfPositionToBeMuted ==  0);
193     BOOST_CHECK(!aArg.isAMutationEventPossible);
194
195     numberOfMutantLocus = {0,0,0,0,0};
196     BOOST_CHECK(aArg.numberOfMutantSNPsByPosition == numberOfMutantLocus );
197
198     leafReachTemp = {0,1,2,3,4,5,6,7};
199     BOOST_CHECK(aArg.getLeafReachabilityforALocus(0,31) == leafReachTemp );
200     BOOST_CHECK(aArg.getLeafReachabilityforALocus(1,31) == leafReachTemp );
201     BOOST_CHECK(aArg.getLeafReachabilityforALocus(2,31) == leafReachTemp );
202     BOOST_CHECK(aArg.getLeafReachabilityforALocus(3,31) == leafReachTemp );
203     BOOST_CHECK(aArg.getLeafReachabilityforALocus(4,31) == leafReachTemp );
204
205     //
206     BOOST_CHECK(aArg.numberOfCoalescencePossible ==  0);
207     coal = {{0}};
208
209     // BOOST_CHECK(aArg.coalescencePossibilitiesForLivesNodes == coal);
210
211     aliveNodes = {31};
212     BOOST_CHECK(aArg.positionOfAliveNodes == aliveNodes );
213
214     }
```

ANNEXE D

CODE MAITRE DE LA FONCTION PARALLÈLE SUR GPU

FastARGBoostTests.cpp

```
216  void BitwiseOperationsOnGPU(){
217      auto startTime = Clock::now(); auto endTime = Clock::now();
218
219      // File stream
220      std::ofstream outfile; outfile.open("CoalescenceTime.dat");
221      outfile << "This file give the time to coalsecence sequences on a GPU versus a
             CPU." << endl;
222      outfile << "Number of snips   GPU    CPU  (time in ms)" << endl;
223
224      // console stream
225      cout << endl;
226      cout << "Here we generate different random sequences and we check " << endl ;
227      cout << "with a compatible one if they can coalesce with it." << endl;
228      cout << "Each time we compare the parallel and sequential time." << endl;
229      cout << "Number of snips   CPU    GPU  (time in ms)" << endl;
230
231      // GPU init section
232      cl_int errorCode;
233
234      // Get a platform IDs
235      cl_platform_id plateform;
236      errorCode = clGetPlatformIDs( 1, &plateform, NULL );
237      generateOpenClError (__func__, __LINE__,  errorCode);
238
239      // Get a GPU OpenCL enable device.
240      cl_device_id device;
241      errorCode = clGetDeviceIDs( plateform, CL_DEVICE_TYPE_GPU, 1, &device, NULL );
242      generateOpenClError (__func__, __LINE__,  errorCode);
243
244      // Creates an OpenCL context.
245      cl_context context = clCreateContext( NULL, 1, &device, NULL, NULL, &errorCode
             );
246      generateOpenClError (__func__, __LINE__,  errorCode);
247
248      // Creates a command-queue on a specific device.
249      cl_command_queue queue = clCreateCommandQueue( context, device, 0, &errorCode )
             ;
250      generateOpenClError (__func__, __LINE__,  errorCode);
251
252      // Get the source code of the Kernel from a file.
253      std::ifstream file ( GPU_COAL_KERNEL );
254      string str, sourceKernel;
```

```
255         while(std::getline(file,str)){
256             sourceKernel += str;
257             sourceKernel += "\n";
258         }
259         file.close();
260 
261         // Apparently in C++11 source doesn't need to be freed! If you try it will
262         // crash the program.
263         // stackoverflow.com/AA347949/how-to-convert-a-stdstring-to-const-char-or-char
264         const char* source = sourceKernel.c_str();
265 
266         // Creates a program object for a context.
267         cl_program program = clCreateProgramWithSource( context, 1, (const char**)&
                source, NULL, &errorCode );
268         generateOpenClError (__func__, __LINE__,  errorCode);
269 
270 
271         // Builds (compiles and links) a program executable from the program source or
                binary.
272         errorCode = clBuildProgram( program, 0, NULL, NULL, NULL, NULL );
273         if (errorCode == CL_BUILD_PROGRAM_FAILURE) {
274             // Determine the size of the log
275             size_t log_size;
276             clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL, &
                    log_size);
277 
278             // Allocate memory for the log
279             char *log = (char *) malloc(log_size);
280 
281             // Get the log
282             clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size, log,
                    NULL);
283 
284             // Print the log
285             printf("\n\n%s\n\n", log);
286         }
287         generateOpenClError (__func__, __LINE__,  errorCode);
288 
289         // Creates a kernal object.
290         cl_kernel kernel =  clCreateKernel( program, "GPU_coal", &errorCode );
291         generateOpenClError (__func__, __LINE__,  errorCode);
292         // End of GPU OpenCL init section
293 
294 
295 
296         // We create a matrix of random snip data to be coalesce on the CPU and on the
                GPU
297         for (size_t nbSnips = MIN_NUMBER_OF_SNIPS ; nbSnips <=  MAX_NUMBER_OF_SNIPS ;
                nbSnips *= 2 ){
298             // Specify to the object the number of desired snips
299             SnipSequence::setSizeOfSequence( nbSnips );
300 
301             //
302             std::valarray<SnipSequence> CPU_Result(NB_OF_SEQUENCES_IN_GPU_COAL);
303             std::valarray<bool> assertintergrityPar(NB_OF_SEQUENCES_IN_GPU_COAL);
304             SnipSequence aFistSequence = SnipSequence::generateARandomSnipSequence(
                    nbSnips);
305 
306             // Random test data generation
307             std::valarray<SnipSequence> testData (NB_OF_SEQUENCES_IN_GPU_COAL);
308             for (size_t i  = 0 ; i < NB_OF_SEQUENCES_IN_GPU_COAL ;++i){
309                 testData[i] = SnipSequence::generateARandomSnipSequence(nbSnips);
310             }
```

```
311
312             // Outputs
313             cout << nbSnips << "␣␣";
314             outfile << nbSnips  << "␣␣";
315
316             // CPU section
317             // Start the timer
318             startTime = Clock::now();
319             CPU_Result = coalescenceASequenceOnAnArray(testData,aFistSequence);
320             endTime = Clock::now();
321             auto timeForCurrentTest = std::chrono::duration_cast<std::chrono::
                    milliseconds> (endTime - startTime).count();
322
323             // Outputs
324             cout << timeForCurrentTest << "␣" ;
325             outfile << timeForCurrentTest << "␣␣";
326
327
328             // GPU timed section
329             startTime = Clock::now();
330
331             // Dynamic allocations of float arrays to be multiply
332             size_t numberOfelement = testData[0].getnumberOfBlock();
333             size_t nbOfBlock = numberOfelement*NB_OF_SEQUENCES_IN_GPU_COAL;
334
335             unsigned int * AMask = (unsigned int*)std::malloc(sizeof(unsigned int) *
                    numberOfelement*NB_OF_SEQUENCES_IN_GPU_COAL);
336             unsigned int * AData = (unsigned int*)std::malloc(sizeof(unsigned int) *
                    numberOfelement*NB_OF_SEQUENCES_IN_GPU_COAL);
337             unsigned int * BMask = (unsigned int*)std::malloc(sizeof(unsigned int) *
                    numberOfelement*NB_OF_SEQUENCES_IN_GPU_COAL);
338             unsigned int * BData = (unsigned int*)std::malloc(sizeof(unsigned int) *
                    numberOfelement*NB_OF_SEQUENCES_IN_GPU_COAL);
339             // cout << "Data in A " << endl;
340             for (size_t i = 0 ; i < testData.size() ; ++i ){
341                 for (size_t j = 0 ; j < numberOfelement ; ++j ){
342                     AMask[i*numberOfelement +j ] = testData[i].AncestralDataMask[j];
343                     AData[i*numberOfelement +j ] = testData[i].snipsData[j];
344                     //  cout << i*numberOfelement +j << " "<<  testData[i].
                            AncestralDataMask[j] << " " <<  testData[i].snipsData[j] <<
                            endl;
345
346                 }
347             }
348             //  cout << "Data in B " << endl;
349             for (size_t i = 0 ; i < testData.size() ; ++i ){
350                 for (size_t j = 0 ; j < numberOfelement ;++j){
351                     BMask[i*numberOfelement +j] = aFistSequence.AncestralDataMask[j];
352                     BData[i*numberOfelement +j] = aFistSequence.snipsData[j];
353                     //  cout << aFistSequence.AncestralDataMask[j]  << " " <<
                            aFistSequence.snipsData[j] << endl;
354                 }
355             }
356
357
358             // Creates buffers for the GPU communications
359             cl_mem buffer_A = clCreateBuffer( context, CL_MEM_READ_ONLY |
                    CL_MEM_COPY_HOST_PTR, sizeof(unsigned int)*nbOfBlock, AMask, &errorCode
                    );
360             generateOpenClError (__func__, __LINE__,  errorCode);
361             cl_mem buffer_B = clCreateBuffer( context, CL_MEM_READ_ONLY |
                    CL_MEM_COPY_HOST_PTR, sizeof(unsigned int)*nbOfBlock, AData, &errorCode
                    );
```

```
362             generateOpenClError (__func__, __LINE__,  errorCode);
363             cl_mem buffer_C = clCreateBuffer( context, CL_MEM_READ_ONLY |
                    CL_MEM_COPY_HOST_PTR, sizeof(unsigned int)*nbOfBlock, BMask, &errorCode
                    );
364             generateOpenClError (__func__, __LINE__,  errorCode);
365             cl_mem buffer_D = clCreateBuffer( context, CL_MEM_READ_ONLY |
                    CL_MEM_COPY_HOST_PTR, sizeof(unsigned int)*nbOfBlock, BData, &errorCode
                    );
366             generateOpenClError (__func__, __LINE__,  errorCode);
367             cl_mem buffer_ResultMask = clCreateBuffer( context, CL_MEM_WRITE_ONLY,
                    sizeof(unsigned int)*nbOfBlock, NULL, &errorCode );
368             generateOpenClError (__func__, __LINE__,  errorCode);
369             cl_mem buffer_ResultData = clCreateBuffer( context, CL_MEM_WRITE_ONLY,
                    sizeof(unsigned int)*nbOfBlock, NULL, &errorCode );
370             generateOpenClError (__func__, __LINE__,  errorCode);
371
372
373             // Configurations of the arguments of the kernel
374             errorCode = clSetKernelArg( kernel, 0, sizeof(cl_mem), &buffer_A );
375             generateOpenClError (__func__, __LINE__,  errorCode);
376             errorCode = clSetKernelArg( kernel, 1, sizeof(cl_mem), &buffer_B );
377             generateOpenClError (__func__, __LINE__,  errorCode);
378             errorCode = clSetKernelArg( kernel, 2, sizeof(cl_mem), &buffer_C );
379             generateOpenClError (__func__, __LINE__,  errorCode);
380             errorCode = clSetKernelArg( kernel, 3, sizeof(cl_mem), &buffer_D );
381             generateOpenClError (__func__, __LINE__,  errorCode);
382             errorCode = clSetKernelArg( kernel, 4, sizeof(cl_mem), &buffer_ResultMask )
                    ;
383             generateOpenClError (__func__, __LINE__,  errorCode);
384             errorCode = clSetKernelArg( kernel, 5, sizeof(cl_mem), &buffer_ResultData )
                    ;
385             generateOpenClError (__func__, __LINE__,  errorCode);
386             unsigned int N = NB_OF_SEQUENCES_IN_GPU_COAL;
387             errorCode = clSetKernelArg( kernel, 6, sizeof(int), &N );
388             generateOpenClError (__func__, __LINE__,  errorCode);
389
390
391             // We send the task to the GPU
392             size_t nb_of_tasks = numberOfelement*NB_OF_SEQUENCES_IN_GPU_COAL;
393
394             cl_event event;
395             errorCode =  clEnqueueNDRangeKernel( queue, kernel, 1, NULL, &nb_of_tasks,
                    NULL, 0, NULL, &event );
396             generateOpenClError (__func__, __LINE__,  errorCode);
397             // Wait until it finishes
398             clWaitForEvents(1, &event);
399
400             // Retrieves the results
401
402             unsigned int * ResultMask = (unsigned int*)std::malloc(sizeof(unsigned int)
                    * nbOfBlock);
403             unsigned int * ResultData = (unsigned int*)std::malloc(sizeof(unsigned int)
                    * nbOfBlock);
404
405             clEnqueueReadBuffer( queue, buffer_ResultMask, CL_TRUE, 0, sizeof(unsigned
                    int)*nbOfBlock, ResultMask, 0, NULL, &event );
406             // Wait until it finishes
407             clWaitForEvents(1, &event);
408             clEnqueueReadBuffer( queue, buffer_ResultData, CL_TRUE, 0, sizeof(unsigned
                    int)*nbOfBlock, ResultData, 0, NULL, &event );
409             // Wait until it finishes
410             clWaitForEvents(1, &event);
411
```

```
412             // Outputs of for the time it took
413             endTime = Clock::now();
414             timeForCurrentTest = std::chrono::duration_cast<std::chrono::milliseconds>
                    (endTime - startTime).count();
415             cout << timeForCurrentTest << "\n" ;
416             outfile << timeForCurrentTest << "\n";
417
418             free(AMask);
419             free(BMask);
420             free(AData);
421             free(BData);
422             free(ResultMask);
423             free(ResultData);
424
425             clReleaseMemObject( buffer_A );
426             clReleaseMemObject( buffer_B );
427             clReleaseMemObject( buffer_C );
428             clReleaseMemObject( buffer_D );
429             clReleaseMemObject( buffer_ResultMask );
430             clReleaseMemObject( buffer_ResultData );
431
432         }// End of for loop that change the number of snips
433
434         outfile.close();
435         clFlush(queue);
436         clFinish(queue);
437         clReleaseKernel( kernel );
438         clReleaseProgram( program );
439         clReleaseCommandQueue( queue );
440         clReleaseContext( context );
441     }
```

RÉFÉRENCES

Bush, W. S. et Moore, J. H. (2012). Chapter 11 : Genome-wide association studies. *PLOS Computational Biology*, *8*(12), 1–11. `http://dx.doi.org/10.1371/journal.pcbi.1002822`. Récupéré de `https://doi.org/10.1371/journal.pcbi.1002822`

Center, G. S. L. (2014). Genetic linkage. Récupéré de `http://learn.genetics.utah.edu/content/pigeons/geneticlinkage/`

Chan, W. F., Gurnot, C., Montine, T. J., Sonnen, J. A., Guthrie, K. A. et Nelson, J. L. (2012). Male microchimerism in the human female brain. *PLoS One*, *7*(9), e45592.

Darwin, C. (1859). On the origin of the species by natural selection.

Fearnhead, P. et Donnelly, P. (2001). Estimating recombination rates from population genetic data. *Genetics*, *159*(3), 1299–1318.

Fisher, R. A. (1930). *The genetical theory of natural selection : a complete variorum edition*. Oxford University Press.

Griffiths, R. C. et Marjoram, P. (1996). Ancestral inference from samples of dna sequences with recombination. *Journal of Computational Biology*, *3*(4), 479–502.

Jukes, T. H., Cantor, C. R. *et al.* (1969). Evolution of protein molecules. *Mammalian protein metabolism*, *3*(21), 132.

Kimura, M. (1980). A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *Journal of molecular evolution*, *16*(2), 111–120.

Lyngsø, R. B., Song, Y. S. et Hein, J. (2005). Minimum recombination histories by branch and bound. Dans *WABI*, 239–250. Springer.

Mader, S. S. (2005). *Biology - Reinforced Nasta Binding for Secondary Market*. W C B/McGraw-Hill. Récupéré de `https://www.amazon.com/Biology-Reinforced-Binding-Secondary-Market/dp/0073258393?`

SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0073258393

Minichiello, M. J. et Durbin, R. (2006). Mapping trait loci by use of inferred ancestral recombination graphs. *Am J Hum Genet*, *79*(5), 910–922. 43640[PII]. Récupéré de http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1698562/

Myers, S. R. et Griffiths, R. C. (2003). Bounds on the minimum number of recombination events in a sample history. *Genetics*, *163*(1), 375–394.

Nature (2001). A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. *Nature*, *409*(6822), 928–933. http://dx.doi.org/10.1038/35057149. Récupéré de http://dx.doi.org/10.1038/35057149

Nature (2007). A second generation human haplotype map of over 3.1 million snps. *Nature*, *449*(7164), 851–861. http://dx.doi.org/10.1038/nature06258. Récupéré de http://dx.doi.org/10.1038/nature06258

Nature (2010). A map of human genome variation from population-scale sequencing. *Nature*, *467*(7319), 1061–1073. http://dx.doi.org/10.1038/nature09534. Récupéré de http://dx.doi.org/10.1038/nature09534

Nguyen, T. T. P., Le, V. S., Ho, H. B. et Le, Q. S. (2017). Building ancestral recombination graphs for whole genomes. *IEEE/ACM transactions on computational biology and bioinformatics*, *14*(2), 478–483.

Song, Y. S. et Hein, J. (2005). Constructing minimal ancestral recombination graphs. *Journal of Computational Biology*, *12*(2), 147–169.

Stephens, M. et Donnelly, P. (2000). Inference in molecular population genetics. *Journal of the Royal Statistical Society : Series B (Statistical Methodology)*, *62*(4), 605–635. http://dx.doi.org/10.1111/1467-9868.00254. Récupéré de http://dx.doi.org/10.1111/1467-9868.00254

Wright, S. (1931). Evolution in mendelian populations. *Genetics*, *16*(2), 97–159.