Académie de Montpellier
Université Montpellier II
Sciences et Techniques du Languedoc

# MÉMOIRE DE STAGE DE MASTER M2

effectué au Laboratoire d'Informatique de Robotique
et de Micro-Électronique de Montpellier

Spécialité : **Professionelle et Recherche unifiée en Informatique**

## Comparaison de génomes et distance de transformation

par **Benjamin SEVERAC**

Date de soutenance : **16 juin 2008**

Sous la direction
de **Sèverine BÉRARD** et **Eric RIVALS**

**Résumé**

Comparer des génomes dans leur globalité est un travail courant en bioinformatique. Les précédentes méthodes d'alignement, ne sont pas applicables à des séquences trop longues pour des raisons de temps de calcul d'une part, mais aussi car elles sont basées sur les opérations de mutations seulement ponctuelles (substitution, insertion, délétion d'un nucléotide). Elles ne peuvent donc pas décrire toute l'information contenue dans les génomes entiers. Pour cela, on doit prendre en compte des événements évolutifs plus nombreux et plus complexes que les seules mutations ponctuelles. Plusieurs approches pour résoudre ce problème ont été développées. L'une d'entre elle est la *distance de transformation* définie par Jean-Stéphane Varré, Jean-Paul Delahaye et Eric Rivals [7]. L'idée de ce stage est de faire évoluer la distance de transformation. Après avoir analysé l'implémentation d'algorithmes de calcul de cette distance. Nous avons développé de nouvelles heuristiques permettant un calcul plus rapide de cette distance et, nous avons amélioré le modèle initial de la distance de transformation en ajoutant la gestion des chevauchements, et en ajoutant la possibilité de prendre en compte les auto-copies qui sont d'autre types d'événements évolutifs.

**Abstract**

Comparing genomes as a whole is a classic work in bioinformatics. The previous alignment methods are not applicable to sequence too long for reasons of computing time, but also because they are based on operations only point mutations (substitution, insertion, deletion of a nucleotide). They can not therefore not describe any information contained in the entire genomes. For this, we must take account of events evolving more and more complex than the only point mutations. Several approaches solving the problem will have been developed. One of them is the *transformation distance* defined by Jean-Stéphane Varré, Jean-Paul Delahaye and Eric Rivals [7]. The idea of this course is to change the distance of transformation. After analyzing the implementation of Jean-Stéphane Varré, algorithms. We develop new heuristics allowing a fastest computing time, and we have improved the initial models of this distance by adding management overlaps, and adding the possibility of taking into account the self-copies.

# Remerciements

Je tiens à remercier tout d'abord mes deux encadrants, Sèverine BÉRARD et Eric RIVALS. Qui m'ont assistés tout au long du stage et qui m'ont fait découvrir la bioinformatique.

Je remercie également tous les membres de la Halles 3 pour leur accueil, et les bons moments partagés. La machine à café sans qui les matins auraient été dur.

Je ne peux oublier les amis et la famille qui sont une grande source d'énergie.

# Table des matières

# 1 Introduction

Depuis la découverte de la structure ADN par Watson et Cricks dans les années 50, l'analyse de ces séquences a été un enjeu primordial pour les biologistes. Avec les projets de séquençages systématiques visant à obtenir la séquence complète de tous les chromosomes d'un organisme, le nombre de données disponibles est en constante augmentation et ce, de façon exponentielle. Le traitement de cette masse de données implique une automatisation des tâches et donc une forte collaboration entre biologistes et informaticiens.

Une approche classique d'analyse de nouvelles séquences ADN est de les comparer avec celles déjà connues. Avec comme hypothèse que si deux séquences se ressemblent, elles ont de fortes probabilités de coder les mêmes fonctions. La comparaison de séquences est donc un problème important en bioinformatique. La méthode classique pour effectuer des comparaisons entre séquences est l'alignement. Cette méthode donne de bons résultats quand les séquences sont proches, mais ne permet pas une bonne détection de similarités quand les séquences sont éloignées. Les alignements procèdent par des comparaisons ponctuelles alors que des comparaisons par blocs similaires reflètent mieux la réalité biologique. Le mode de comparaison des alignements respecte l'ordre de lecture des nucléotides dans les séquences or, certaines séquences évoluent en déplaçant ou en dupliquant du matériel génétique, ainsi l'ordre des nucléotides n'est pas conservé. D'autres modes de comparaison ont été inventés pour remédier à ces défauts, l'un d'eux, *la distance de transformation* [7] permet la comparaison de séquences, grâce à la détection de segments communs, et en autorisant ceux-ci à ne pas être dans le même ordre dans les séquences.

La *complexité de Kolmogorov* [5] mesure la taille de la plus petite description d'un objet. La complexité relative de Kolmogorov mesure la taille de la plus petite description d'un objet relativement à un autre. Elle permet ainsi la mesure de la quantité d'information qui doit être ajoutée à la description du premier objet pour obtenir le second.

La complexité de Kolmogorov n'est pas calculable, mais des *algorithmes de compression* permettent d'en obtenir une majoration. En effet, un algorithme de compression permet de décrire un objet en minimisant sa taille finale. On tente donc d'obtenir la plus petite description de cet objet. Malheureusement, un algorithme de compression suit des règles qui ne permettent pas de prouver que la description de l'objet soit la plus compacte possible. La compression d'un objet n'est donc qu'une approximation de la mesure de complexité de Kolmogorov.

La *distance de transformation* mesure la quantité minimale d'information qui doit être ajoutée à une séquence source $\mathcal{S}$ pour obtenir une séquence cible $\mathcal{C}$. Cette distance est obtenue avec la mesure de la longueur d'un *script*, qui est une suite d'opérations qui permet d'obtenir la séquence cible grâce à la source. Les opérations sont de deux types les copies, qui copient un segment de la séquence source pour formé la cible, et les insertions qui insèrent un segment non présent dans la séquence source.

Ce mémoire porte sur l'étude de la *distance de transformation* afin d'en améliorer le calcul. Les différentes voies explorées durant le stage ont permis d'améliorer les temps de calcul, mais aussi d'enrichir le modèle en y ajoutant par exemple la gestion des chevauchements et les auto-copies.

Dans une première partie, nous présentons les méthodes existantes et les outils qui nous sont utiles. Par la suite, nous verrons le travail effectué pendant ce stage : les améliorations apportées tant au niveau algorithmique qu'au niveau du modèle et leurs implémentations.
Dans la dernière partie, nous comparerons nos travaux à ceux déjà existants.

# 2 État de l'art

Dans cette section nous présentons les différents éléments que nous avons étudié au cours de ce stage.

**Définition 1** *Une séquence est un mot de taille $l$, ces lettres sont numéroté de $0$ à $l-1$.*

**Notation 1** *Nous notons $\mathcal{S}$ la séquence source, et $\mathcal{C}$ la séquence cible.*

**Définition 2** *Un segment est un intervale dans une séquence.*

**Notation 2** *Nous nommons* appariement *une paire de segments similaires entre deux séquences.*

## 2.1 Algorithme de recherche de segments similaires

Pour calculer la distance de transformation entre deux séquences, nous avons besoin de connaître les segments communs entre ces deux séquences. Dans cette section nous allons présenter brièvement deux algorithmes permettant de trouver des segments similaires entre deux séquences.

**Définition 3** *Un* appariement *est défini par un* triplet $(p, q, l)$*, où $p$ est l'indice de début dans la séquence* cible*, $q$ est l'indice de début dans séquence* source *et $l$ la longueur du segment similaire.*

**Définition 4** *Soit* 2 *appariements : $f = (p, q, l)$ et $g = (p', q', l')$. On définit l'inclusion de $f$ dans $g$, notée : $f \subseteq g$, ssi :*
- $p' \leq p$
- $q' \leq q$
- $q + l \leq q' + l'$
- $p + l \leq p' + l'$

Un exemple d'inclusion est illustré figure 1. L'appariement $f = (3, 1, 4)$ représentant le segment similaire TGAC, est inclus dans l'appariement $g = (2, 0, 6)$ représentant le segment similaire ATGACC.

Nous présentons dans la suite l'algorithme de Leung et al. [4], utilisé dans la méthode décrite dans [7], puis l'algorithme de Vmatch [2], que nous avons utilisé dans notre implémentation pour améliorer les performances de la méthode.

Les algorithmes de Vmatch et Leung recherchent les segments communs maximaux entre deux séquences. C'est-à-dire que si $f$ est un appariement de l'ensemble $\mathcal{F}$ des appariements retournés alors $\forall g \in \mathcal{F}$, $g \neq f \Rightarrow f \nsubseteq g$.

### 2.1.1 Leung et al.

Leung permet de trouver des paires de segments exacts ou des paires de segments approximés. L'approximation permet l'ajout de substitutions dans les paires de segments similaires. L'algorithme permet de paramétrer le nombre maximal consécutif de substitutions autorisées (si on n'autorise aucune substitution, on obtient les appariements exacts). On peut également choisir la taille minimale des segments similaires, en effet les segments trop petits ne sont pas significatifs (trivialement les segments de taille 1). La figure 1 montre un exemple d'appariement exact, et la figure 2 montre un exemple d'appariement approximé avec 2 substitutions consécutives autorisées.

| séquence 1 | T | C | A | **T** | **G** | **A** | **C** | C | T | A |
|---|---|---|---|---|---|---|---|---|---|---|
| séquence 2 | A | **T** | **G** | **A** | **C** | C | C | T | T | A |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

FIG. 1 – Appariement exact. Les cases grisées correspondent aux nucléotides de l'appariement $g = (2, 0, 6)$, Les caractères en gras correspondent aux nucléotides de l'appariement $f = (3, 1, 4)$. $f \subseteq g$

| séquence 1 | T | C | A | T | G | C | C | C | C | A |
|---|---|---|---|---|---|---|---|---|---|---|
| séquence 2 | A | T | C | A | C | A | C | T | T | A |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

FIG. 2 – Appariement approximé. Les cases grisées correspondent aux nucléotides de l'appariement $(2, 0, 7)$ qui est approximé. Les cases foncées correspondent aux substitutions.

### 2.1.2 Vmatch

Vmatch permet de trouver des paires de segments exacts et des paires de segments inversés et complémentés.

**Définition 5** *Les appariements inversés et complémentés sont définis ainsi :*
- *Un des deux segments est inversé (ex. ATCC devient CCTA)*
- *On complémente chaque nucléotide de la séquence inversé cela revient à appliquer les règles suivantes $A \to T$, $T \to A$, $G \to C$, $C \to G$.*

Sur figure 3, CGTTACGA et TCGTAACG forment un appariement inversé et complémenté. On s'intéresse à ce type de segment, car ils sont biologiquement équivalents. En effet, la molécule d'ADN comporte deux brins qui codent la même information génétique. Ces deux brins sont disposés côte à côte sur une double hélice, et l'un des brins est l'inverse et le complément de l'autre. Certains réarrangements génomiques existent entre ces deux brins, il faut donc intégrer ce genre de réarrangements dans le modèle.

| séquence 1 | T | C | G | T | T | A | C | G | A | T |
|---|---|---|---|---|---|---|---|---|---|---|
| séquence 2 | T | A | T | C | G | T | A | A | C | G |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

FIG. 3 – Appariement inversé et complémenté. Les cases grisées correspondent aux nucléotides de l'appariement $(1, 2, 8)$ qui est inversé et complémenté.

## 2.2 Article Varré et al

L'article [7] présente une méthode pour calculer la *distance de transformation*. Si on considère deux séquences : la source $\mathcal{S}$, et la cible $\mathcal{C}$, cette distance consiste à construire la cible $\mathcal{C}$ grâce à la source $\mathcal{S}$. Cette construction se fait en suivant une liste d'opérations qui construit la cible itérativement de gauche à droite ; cette liste est appelée *script*. Les opérations ont un poids, le poids d'un script est défini comme la somme des poids des opérations qui le composent. La distance de transformation est le poids d'un script de poids minimum. Les auteurs proposent un ensemble d'opérations possible pour construire $\mathcal{C}$ en fonction de $\mathcal{S}$, et pour chacune de ces opérations, ils définissent également un poids.

### 2.2.1 Opérations

Nous présentons maintenant les trois opérations définies dans [7] et leur poids :

– La *copie* prend un segment de la séquence source et le copie pour former un segment de la séquence cible. Voir exemple figure 4.
– La *copie inversée et complémentée* prend un segment de la séquence source et le copie en l'inversant et le complémente pour former un segment de la séquence cible. Par simplicité on la nomme *copie inversée*. Voir exemple figure 5.
– *L'insertion* qui permet de créer les zones de la séquence cible qui ne sont pas contenues dans la séquence source.

Ces opérations ont été choisies de manière à simuler les événements biologiques. En effet au cours de l'évolution, les génomes sont soumis à des transformations. Ces opérations permettent de simuler le mécanisme des transformations les plus fréquentes.

**Notation 3** *Pour un appariement $f$ nous noterons $f.p$ le début du segment dans la séquence cible, $f.q$ le début du segment dans la séquence source, et $f.l$ sa longueur.*

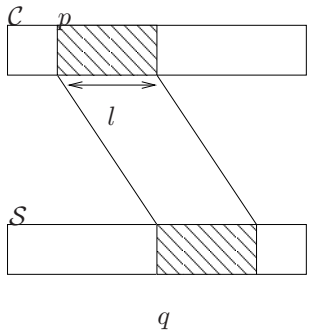


FIG. 4 – Copie.

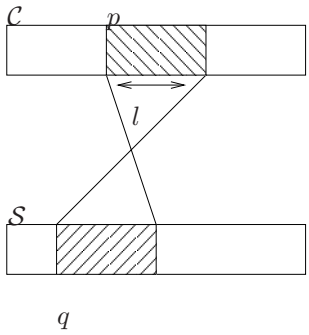


FIG. 5 – Copie inversée complémentée.

### 2.2.2 Poids des opérations

Le poids d'une opération correspond au nombre de bits nécessaires pour encoder celle-ci, ainsi chaque opération est définie par une série de bits et ceux de façon à ce que le codage soit auto-délimité. Nous rappelons que le codage d'un entier $i$ requiert en binaire au minimum $\lceil log_2(i) \rceil$ bits. Les premiers bits de la série nous permettent de connaître le type de l'opération. Nous disposons de 3 opérations. Ainsi pour encoder le type de l'opération il faut $\lceil log_2(3) \rceil$ bits. Les autres bits définissent les paramètres de l'opération. En voici le détail pour chaque opération :

**Copie et copie-inversée** Pour effectuer une copie qu'elle soit inversée ou non, il faut 3 informations : la longueur du segment, le décalage entre la position du segment dans $\mathcal{S}$ et dans $\mathcal{C}$, ainsi que le signe du décalage (+ ou -). Il faut donc coder en binaire ces 3 informations. Le poids de la copie pour un appariement $f$, notée $\omega copie(f)$, a donc un poids total de :

$$\omega copie(f) = \lceil log_2(3) \rceil + \lceil log_2(f.l) \rceil + \lceil log_2(|f.p - f.q|)) \rceil + 1$$

**Insertion** Pour effectuer une insertion les informations à coder sont : le mot à insérer et sa longueur. L'alphabet comporte 4 lettres ACGT, il faut donc $\log_2(4) = 2$ bits par lettre et de $2 \times |m|$ bits pour coder un mot $m$. Le poids de l'insertion d'un mot $m$ noté $\omega insertion(mot)$ a donc un poids total de :

$$\omega insertion(m) = \lceil log_2(3) \rceil + \lceil log_2(|m|) \rceil + 2|m|$$

### 2.2.3 Recherche des segments similaires

Le problème est de reconstruire la cible grâce à des segments de la source, il faut donc trouver les segments communs entre les deux séquences. Dans l'article [7] l'algorithme utilisé pour récupérer les paires de segments similaires est l'algorithme de Leung.

### 2.2.4 Obtenir le script de poids minimal

Nous avons défini les opérations de reconstruction et leurs coûts et nous avons un algorithme permettant de trouver les appariements. Nous allons décrire dans cette partie comment trouver le script de poids minimum. Pour cela, nous introduisons des définitions supplémentaires.

**Définition 6** *Soit deux appariements $f$ et $g$, $f$* précède *$g$, noté $f < g$, ssi $f.p + f.l \leq g.p$. Cette relation est illustrée sur la figure 6.*

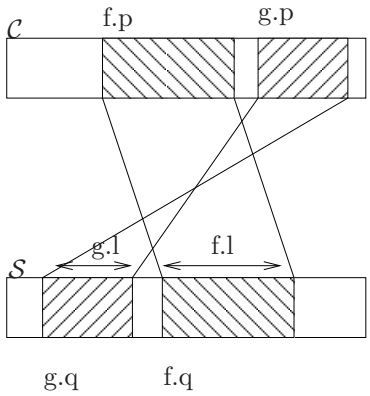


FIG. 6 – Ordre sur les appariements. Soit 2 appariements $f$ et $g$ avec $f < g$

Soit $\mathcal{F}$ l'ensemble des appariements trouvé de $\mathcal{S}$ à $\mathcal{C}$. Les auteurs définissent un graphe orienté acyclique (GOA ou DAG en anglais) $G$ nommé *script-graph* dont les arcs et les sommets sont pondérés. Les sommets représentent les appariements de $\mathcal{S}$ à $\mathcal{C}$, et les arcs les insertions. Ce graphe possède deux sommets particuliers : $s$ la source et $w$ le puits. Un chemin $c$ de $s$ à $p$ dans ce graphe représente un script $k$ qui construit $\mathcal{C}$ à partir de $\mathcal{S}$. La valuation de $c$ est la somme du poids des sommets et des arcs qu'il contient. Cette valuation est égale au coût du script $k$.

**Définition 7** *Construction du script-graph entre $\mathcal{S}$ et $\mathcal{C}$, $G = (V, E)$ obtenu de la manière suivante :*

- *$V = \mathcal{F} \cup \{s, w\}$ ou $s$ la source, $w$ le puits.*
- *soit $f, g \in \mathcal{F}$*
  - *$(f, g) \in E$ ssi $f < g$*

Le poids d'un noeud correspond au poids de l'appariement (le poids des sommets $s$ et $w$ est nul) qu'il représente et le poids d'une arête $(i, j)$ correspond au poids de l'insertion entre les appariements $i$ et $j$. Trouver le script de poids minimal revient à trouver le chemin de poids minimal entre la source et le puits dans le script-graph. Le graphe étant un DAG, la complexité en temps pour trouver un chemin de poids minimal est $O(|V| + |E|)$.

**Remarque** La propriété de construction est exprimée de manière légèrement différente dans l'article [7]. Mais c'est elle qui est utilisée dans l'implémentassion de J.S.Varré, c'est pour cela que nous avons fait le choix de présenter celle-ci pour plus de clarté.

### 2.2.5 Exemple récapitulatif

Cette section a pour but de présenter un exemple complet de toutes les étapes vues dans les sections précédentes. La figure 7 montre deux séquences $\mathcal{C}$ et $\mathcal{S}$. Le problème est donc de trouver le meilleur script qui construit $\mathcal{C}$ en fonction de $\mathcal{S}$. La méthode présentée recherche les appariements entre ces deux séquences par l'algorithme de Leung avec une taille minimum pour les appariements de 7. Ici nous avons pris les appariments *maximaux* :

1. AACTTTGCTACT défini par le triplet (5,9,12)
2. CTACTCCTACTAGGGCAA défini par le triplet (12,23,18)
3. ACTGCAT défini par le triplet (33,0,7)

| | |
|---|---|
| $\mathcal{C}$ | GCGAG**AACTTTG*CTACT***$CCTACTAGGGCAA$CCC**ACTGCAT**CCTACG |
| $\mathcal{S}$ | **ACTGCAT**AA**AACTTTGCTACT**AA$CTACTCCTACTAGGGCAA$ |

FIG. 7 – Exemple de 2 séquences ADN avec leurs appariements

Le script-graph $G$ est créé comme indiqué par la propriété 7. Trois sommets sont donc créés, représentant chacun un des appariements on rajoute deux sommets $s$ et $w$. Ensuite les arcs sont ajoutés selon la propriété 7. Il faut également calculer le poids des sommets et des arcs, en voici le détail :

- $\omega copie(1) = \lceil log_2(3) \rceil + \lceil log_2(12) \rceil + \lceil log_2(|5-9|) \rceil + 1 = 2+4+2+1 = 9$
- $\omega copie(2) = \lceil log_2(3) \rceil + \lceil log_2(18) \rceil + \lceil log_2(|12-23|) \rceil + 1 = 2+5+4+1 = 12$
- $\omega copie(3) = \lceil log_2(3) \rceil + \lceil log_2(7) \rceil + \lceil log_2(|33-0|) \rceil + 1 = 2+3+6+1 = 12$

La notation $\omega insertion(x, y)$ correspond au poids de l'arc $(x, y)$, i.e., l'insertion entre les appariements $x$ et $y$ dans la cible.

- $\omega insertion(s, 1) = \lceil log_2(3) \rceil + \lceil log_2(|GCGAG|) \rceil + 2|GCGAG| = 2+3+10 = 15$
- $\omega insertion(s, 2) = 2 + \lceil log_2(12) \rceil + 12 * 2 = 30$
- $\omega insertion(s, 3) = 2 + \lceil log_2(33) \rceil + 33 * 2 = 74$
- $\omega insertion(s, w) = 2 + \lceil log_2(46) \rceil + 46 * 2 = 100$
- $\omega insertion(1, 3) = 2 + \lceil log_2(16) \rceil + 16 * 2 = 38$
- $\omega insertion(1, w) = 2 + \lceil log_2(29) \rceil + 29 * 2 = 65$
- $\omega insertion(2, 3) = 2 + \lceil log_2(3) \rceil + 3 * 2 = 10$
- $\omega insertion(2, w) = 2 + \lceil log_2(16) \rceil + 16 * 2 = 38$
- $\omega insertion(3, w) = 2 + \lceil log_2(6) \rceil + 6 * 2 = 17$

La figure 8 représente le script-graph $G$. Le calcul d'un chemin de poids minimal dans le script-graph nous donne le chemin en gras, son poids est de 80. Une fois le chemin de poids minimal calculé, nous pouvons créer le script de poids minimal qui construit $\mathcal{C}$ :

- *insertion*(GCGAG)
- *copie*(1)
- *insertion*(CCTACTAGGGCAACCCACTGCATACTACG)

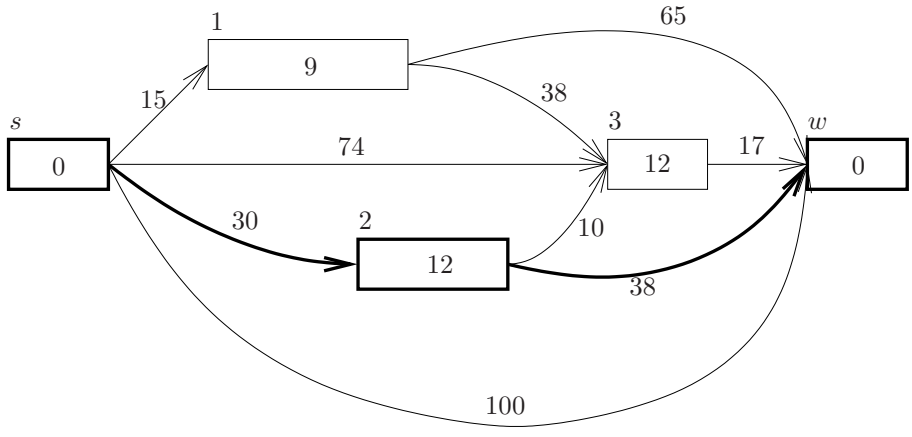


FIG. 8 – script-graph $G$. La valuation sur les arcs correspond au poids des insertions. La valuation à l'intérieur des rectangles est le poids des copies. Les caractères au dessus des rectangles sont les noms des sommets. Le chemin en gras correspond au script de poids minimal

# 3 Réalisation

Cette section présente le travail effectué au cours du stage.

## 3.1 Analyse de l'existant

Dans un premier temps, nous avons étudié l'implémentation du calcul de la distance de transformation, fournie par J.S. Varré l'un des auteurs de l'article [7]. Son implémentation est codée en $C$, et comprend divers outils et algorithmes. Cette analyse a fait ressortir divers points :

- L'implémentation comprend l'algorithme de Leung qui s'avère ne pas prendre en charge les appariements inversés et complémentés.
- Nous avons vu dans la section précédente une méthode pour calculer le coût des opérations. Dans l'implémentation de J.S. Varré d'autres méthodes sont proposées en addition de celles de l'article. Ces méthodes calculent également le nombre de bits nécessaire pour coder les opérations, mais elles codent les informations nécessaires différemment.
- Trois algorithmes sont implémentés. Le premier est l'algorithme exact, et les deux autres sont des heuristiques. Ces trois algorithmes sont tous basés sur le script-graph introduit dans l'article et présenté dans la section précédente de ce rapport.

Dans cette partie nous allons nous intéresser à deux des algorithmes de l'implémentation de J.S. Varré.

### 3.1.1 Algorithme exact

Cet algorithme trouve le meilleur script, il est dérivé de la définition 7. Ci-dessous la description d'éléments nécessaires pour l'algorithme.

- Soit $\alpha$ un tableau dont les cases d'indice $i$ contiennent le poids du meilleur chemin allant de la source $s$ au sommet $i$.
- Soit $prec$ un tableau dont les cases d'indice $i$ contiennent l'indice du sommet qui précède $i$ sur le chemin de meilleur poids allant de la source au sommet $i$.
- Soit $\omega_i$ une fonction qui donne le poids de l'opération liée au sommet $i$ (copie, copie inversée).
- Soit $\omega_{i,j}$ une fonction qui donne le poids de l'insertion entre $i$ et $j$, i.e., de l'arête $(i, j)$.

Le calcul des éléments de $\alpha$ est donné par la formule de récurrence suivante (avec $\mathcal{F}$ l'ensemble des appariements entre $\mathcal{S}$ et $\mathcal{C}$) :

$$\alpha_i = \forall j \in \mathcal{F} \mid j < i,\ Min(\alpha_j + \omega_{j,i} + \omega_i)$$

L'idée intéressante de cet algorithme est qu'il utilise le principe de programmation dynamique. Ainsi, on ne stocke pas les arcs du graphe. Pour chaque sommet $i$ on calcule $\alpha_i$. Pour cela il suffit de parcourir les sommets $j$ précédant $i$ et de mettre à jour $\alpha_i$. Voici le détail de l'algorithme :

**Algorithme 1** : Exact

**Données** : Une liste $\mathcal{F}$ de $n$ appariements entre $\mathcal{S}$ et $\mathcal{C}$.
**Résultat** : Le script de poids minimal dans le script-graph.
$F.ajout(s)$ `// ajout de la source`
$F.ajout(w)$ `// ajout du puits`
`// tri de` $\mathcal{F}$ `par ordre croissant sur` $p$ `(le début de l'appariement dans` $\mathcal{C}$`)`
$sort(\mathcal{F})$;
initialisation des cases de $\alpha$ à $\infty$;
initialisation des cases de $prec$ à -1;
**pour** $i = 0$ ; $i < \mathcal{F}.size()$ ; $i++$ **faire**
  **pour** $j = 0$ ; $j < i$ ; $j++$ **faire**
    `// mise à jour du poids`
    **si** $\mathcal{F}_j < \mathcal{F}_i$ **alors**
      **si** $\alpha_j + \omega_{j,i} + \omega_i < \alpha_i$ **alors**
        $\alpha_i \leftarrow \alpha_j + \omega_{j,i} + \omega_i$;
        $prec[i] \leftarrow j$;

Cet algorithme a une complexité en temps en $\mathcal{O}(n^2)$ où $n$ est le nombre de sommets du graphe (i.e. le nombre d'appariements). Le problème de l'algorithme est que sur des génomes compléts le nombre trop important de sommets entraîne un temps d'exécution non acceptable. Les auteurs ont donc mis au point des heuristiques afin de réduire le temps d'exécution.

### 3.1.2 Heuristique Valid-Script

L'idée de cette heuristique est de ne pas prendre en compte certain arcs transitif du script-graph et ainsi être plus rapide que l'algorithme exact. Malheureusement le script-graph n'est pas triangulé aussi enlever des arcs ne garantie plus l'optimalité. Le choix des arcs est fait selon la définition suivante :

**Ensemble chevauchant**

**Définition 8** Ensemble chevauchant *: Étant donnés deux appariements $f$ et $g$, si $f$ et $g$ se chevauchent dans la séquence cible on dira alors qu'ils sont dans le même ensemble chevauchant.*

La figure 9 illustre cette définition.

**Note :** Soient trois appariements $f$, $g$ et $h$. Si $f$ ne chevauche pas $h$ ils peuvent néanmoins être dans le même ensemble s'ils chevauchent tous les deux $g$.

**Définition 9** *Soient $f$ et $g$ deux sommets du script-graph. l'arc $(f, g)$ est dit* transitif *si un chemin de longueur au plus* 1 *existe entre $f$ et $g$.*

Un ordre est défini sur les ensembles chevauchants. Il est défini de la manière suivante :

**Définition 10** *Soient $I$ et $J$ deux ensembles chevauchants, on note $I < J$ si $\forall i \in I$ et $\forall j \in J$ alors $i.p + i.l < j.p$.*

**Arcs pris en compte** Nous allons définir $G'$ un sous-ensemble du script-graph $G = (V, E)$ tel que $G' = (V, E')$ avec $E' \subseteq E$ :

1. Soient $f$ et $g$ deux appariements d'un ensemble chevauchant $I$, si $f < g$ alors $(f, g) \in E'$
2. Soient $I$ et $J$ deux ensembles chevauchants, si $I < J$ et $\nexists H$ tel que $I < H < J$ et :
   (a) Soit $i \in I$ tel que $\nexists h \in I$ tel que $i < h$
   (b) Soit $j \in J$ tel que $\nexists h \in J$ tel que $h < j$
   (c) Alors $(i, j) \in E'$.

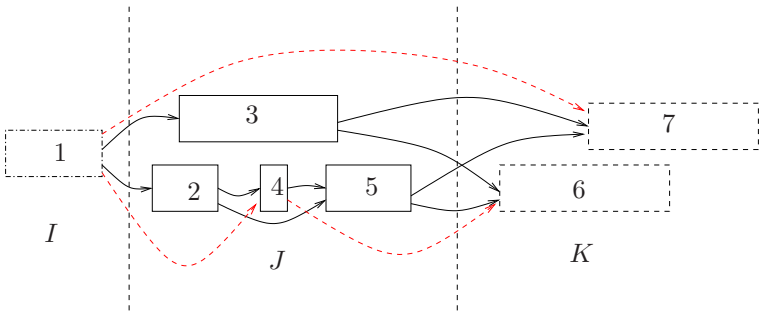


FIG. 9 – *Ensembles chevauchants*. Les rectangles représentent les sommets (i.e. les appariements) du script-graph. Leurs positions et leurs tailles correspondent à leurs positions et longueurs dans la séquence cible. $I = \{1\}, J = \{2, 3, 4, 5\}$ et $K = \{6, 7\}$ sont des ensembles chevauchants. Les arcs pleins sont les arcs de $G'$, ceux en pointillés sont des exemples d'arcs n'appartenant pas à $G'$

Il existe donc deux types d'arcs les arcs *internes* (1) et les arcs *externes* (2). L'idée de l'algorithme est basé sur l'algorithme exact, les structures utilisées sont les mêmes. La différence est qu'il faut contraindre les arcs à $G'$. Voici le détail de l'algorithme :

**Algorithme 2** : valid-script

**Données** : Un ensemble $\mathcal{F}$ de $n$ appariements.
**Résultat** : Le script de poids minimal dans le graphe $G'$.
$\mathcal{F} \leftarrow s$;
$\mathcal{F} \leftarrow p$;
$sort(\mathcal{F})$;
**pour** $j = 0\ ;\ j < \mathcal{F}.size()\ ;\ j++$ **faire**
  $i \leftarrow j - 1$;
  `// Arcs interne`
  `// Linked indique si` $\mathcal{F}[j]$ `possède un arc entrant. quand linked=0 alors (b) est vérifié`
  $linked \leftarrow false$;
  `// beginGroup est le début de l'appariement minimum du groupe de` $j$
  $beginGroup = \mathcal{F}[j].p$;
  `// Quand` $beginGroup < \mathcal{F}[i].p + \mathcal{F}[i].l$ `alors l'appariement` $i$ `fait partie de l'ensemble chevauchant de` $j$
  **tant que** $i \geq 0$ *et* $beginGroup < \mathcal{F}[i].p + \mathcal{F}[i].l$ **faire**
    `// on met à jour l'ensemble chevauchant`
    **si** $beginGroup < \mathcal{F}[i].p$ **alors**
      $beginGroup \leftarrow \mathcal{F}[i].p$;
    `// si un arc interne est possible on le met`
    **si** $\mathcal{F}[j]$ *ne chevauche pas* $\mathcal{F}[i]$ **alors**
      Mise à jour poids de $\alpha_j$ en fonction de l'appariement $\rangle$;
      $linked \leftarrow true$;
    $i--$;
  `// Arcs externe`
  `// limit permet de déterminer quand l'arc` $(i, j)$ `est transitif`
  $limit \leftarrow 0$;
  **si** $linked = 0$ **alors**
    **tant que** $limit \leq \mathcal{F}[i].p + \mathcal{F}[i].l$ **faire**
      **si** $limit < \mathcal{F}[i].p$ **alors**
        $limit \leftarrow \mathcal{F}[i].p$
      Mise à jour poids de $\alpha_j$ en fonction de l'appariement $\rangle$;
      $i--$;

Cette heuristique a une complexité en temps en $\mathcal{O}(nlogn+m)$, $m$ étant le nombre d'arcs de $G'$. Dans le pire des cas il existe qu'un seul ensemble chevauchant, l'algorithme revient donc à l'algorithme exact : étant donné que tous les arcs internes aux ensembles chevauchants sont pris en compte.

## 3.2 Amélioration des algorithmes (temps, heuristique)

Après l'analyse des algorithmes existants, nous avons développé de nouveaux algorithmes permettant de gagner en performance. En effet, l'algorithme exact est trop peu performant pour fonctionner sur des données réelles. Et l'algorithme valid-script est malheureusement trop sensible à la topologie du script-graph : les chevauchements créent énormément d'arcs. Les heuristiques que nous avons développées restent dans l'optique de valid-script c'est-à-dire enlever des arcs transitifs.

### 3.2.1 Heuristique 0-transitif

Cette heuristique étend l'aspect de l'algorithme valid-script en n'autorisant aucun arc transitif dans le graphe. On ne se sert pas de la notion d'ensemble chevauchant. $G''$ ne possède donc plus d'arc transitif ce qui réduit le nombre d'arcs et donc le calcul. Nous définissons $G'' = (V, E'')$ un sous-ensemble du script-graph $G = (V, E)$ tel que $G'' \subseteq G' \subseteq G$

**Définition 11** *Soit* $G'' = (V, E'')$ *obtenu ainsi :*
– $V = \mathcal{F} \cup \{s, w\}$ *ou* $s$ *est la source,* $w$ *le puits.*

- *soit* $f, g \in \mathcal{F}$
  - $(f, g) \in E''$ *ssi* $f < g$ *et* $\nexists\, h \in \mathcal{F},\ f < h < g$
  - $(s, g) \in E''$ *ssi* $\nexists\, h \in \mathcal{F},\ h < g$
  - $(f, w) \in E''$ *ssi* $\nexists\, h \in \mathcal{F},\ f < h$

Les appariements $\mathcal{F}$ sont triés par ordre de fin dans la séquence cible. Ce tri induit certaines propriétés :

**Propriété 1** *Soient $i$ et $j$ deux appariements avec $i \not< j$ alors les appariements $h$ d'indices supérieurs à $i$ dans $\mathcal{F}$ trié vérifient $h \not< j$.*

Soient $i(p, q, l)$, $j(p', q', l')$ et $h(p'', q'', l'')$. On à $j.p < i.p + i.l < h.p + h.l$. On a donc bien $h \not< j$. Sur la figure 10 l'appariement $i$ d'indice 5 et l'appariement $j$ d'indice 7 vérifie $i \not< j$ les appariements $h$ d'indices supérieurs à 5 vérifient donc bien $h \not< j$.

**Propriété 2** *Soient $i$, $j$ deux appariements avec $i < j$ alors les appariements $k$ d'indices inférieurs à $i$ dans $\mathcal{F}$ trié vérifient $k < j$.*

Soient $i(p, q, l)$, $j(p', q', l')$ et $k(p'', q'', l'')$. On à $k.p + k.l < i.p + i.l < j.p$. On a donc bien $k < i$. Sur la figure 10 l'appariement $i$ d'indice 4 et l'appariement $j$ d'indice 7 vérifie $i < j$ les appariements $h$ d'indices inférieurs à 4 vérifient donc bien $h < j$.

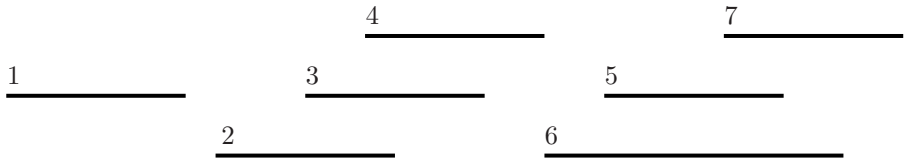


FIG. 10 – Appariements dans la séquence cible. Les numéros des appariements sont les indices de $\mathcal{F}$ trié

Ainsi quand on détermine que l'arc $(i, j)$ est transitif alors tous les arcs $(k, j)$ avec $k$ d'indices inférieurs à $i$ dans $\mathcal{F}$ sont également transitifs.

Des propriétés précédentes on peut déduire que, étant donné $i$ un appariement, les arcs $(k, i)$ non-transitif sont consécutifs dans le tableau $\mathcal{F}$ trié, le calcul en est donc simplifié. Exemple sur la figure 10. Soit $i$ l'appariement d'indice 7 les arcs non transitifs sont bien de 2 à 4 dans le tableau $\mathcal{F}$ trié.

Voici le détail de l'algorithme : La fonction $Nearest(i)$ retourne l'indice dans $\mathcal{F}$ de $j$ le premier appariement non chevauchant de $i$. Cette fonction peut être calculée en $\mathcal{O}(log(n))$ avec une recherche par dichotomie.

**Algorithme 3** : 0-transitif

**Données** : Un ensemble $\mathcal{F}$ de $n$ appariements.
**Résultat** : Le script de poids minimal dans $G''$.

```
F ← s;
F ← w;
sort(F);
pour i = 0 ; i < F.size() ; i++ faire
    j ← Nearest(i);
    limit ← 0;
    tant que limit ≤ F[j].p + F[j].l faire
        si limit < F[j].p alors
            limit ← F[j].p
        // mise à jour du poids
        si α_j + ω_{j,i} + ω_i < α_i alors
            α_i ← α_j + ω_{j,i} + ω_i;
            prec[i] ← j;
        j − −;
```

La complexité en temps de l'algorithme 0-transitif est en $\mathcal{O}(nlogn + m)$, $m$ étant le nombre d'arcs de $G''$.

**Exemple** Nous allons reprendre l'exemple de la section 2.2.5, la figure 11 montre le graphe $G''$ obtenu avec 0-transitif. Il n'y a donc plus d'arcs transitifs. Si on calcule le chemin de poids minimal on obtient le chemin en gras qui donne un script de poids 81 ce qui plus grand que celui trouvé par l'algorithme exact qui est de 80. Ceci illustre bien le fait que le script-graph n'est pas triangulé.

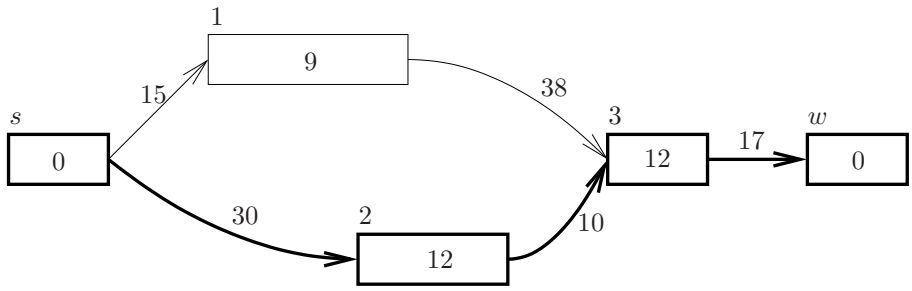


FIG. 11 – graphe $G''$. La valuation sur les arcs correspond au poids des insertions. La valuation à l'intérieur des rectangles est le poids des copies. Les caractères au dessus des rectangles sont les noms des sommets. Le chemin en gras correspond au script de poids minimal

### 3.2.2 Extension de 0-transitif : 1-transitif

Les arcs transitifs apportant parfois de meilleurs chemins, il est dommage de tous les enlever. C'est pourquoi un algorithme permettant d'en garder certains est intéressant. Le problème est de déterminer quels arcs sont susceptibles de réduire le poids du chemin. Faute de trouver avec certitude les arcs transitifs intéressants, nous avons expérimenté plusieurs solutions heuristiques d'ajouts d'arcs transitifs. De ces expérimentations nous avons retenu l'algorithme 1-transitif que nous présentons dans cette section.

L'idée de 1-transitif est d'ajouter pour chaque arc de $G''$ un arc transitif. On se retrouve donc avec $2m$ arcs avec $m$ le nombre d'arcs de $G''$. L'ajout de ces arcs se fait de la manière suivante :

- On parcours les sommets par ordre de fin dans la séquence cible.
- Pour un sommet $j$ on calcule $\alpha_j$ :
  - Pour tous les arcs non-transitif $(i, j)$ avec $h = prec_i$
  - Ajout de l'arc $(h, j)$

Dans l'algorithme 0-transitif lors de l'évaluation d'un arc $(j,i) \in E''$, il est facile de rajouter l'évaluation de l'arc $(h,i)$. En effet, le chemin de poids minimal $k$ a déjà été calculé et l'obtention du sommet $h$ se fait avec le tableau *prec*. Voici le détail de l'algorithme :

**Algorithme 4** : 1-transitif

**Données** : Un ensemble $\mathcal{F}$ de $n$ appariements.
**Résultat** : Le script de poids minimal dans $G''$.

$F \leftarrow w$;
$F \leftarrow p$;
$sort(F)$;
**pour** $i = 0$ ; $i < \mathcal{F}.size()$ ; $i++$ **faire**
  $j \leftarrow Nearest(i)$;
  $limit \leftarrow 0$;
  **tant que** $limit \leq \mathcal{F}[j].p + \mathcal{F}[j].l$ **faire**
    **si** $limit < \mathcal{F}[j].p$ **alors**
      $limit \leftarrow \mathcal{F}[j].p$
    `// mise à jour du poids`
    **si** $\alpha_j + \omega_{j,i} + \omega_i < \alpha_i$ **alors**
      $\alpha_i \leftarrow \alpha_j + \omega_{j,i} + \omega_i$;
      $prec[i] \leftarrow j$;
    `// Ajout d'un arc transitif`
    $h \leftarrow prec[i]$;
    **si** $\alpha_h + \omega_{h,i} + \omega_i < \alpha_i$ **alors**
      $\alpha_i \leftarrow \alpha_h + \omega_{h,i} + \omega_i$;
      $prec[i] \leftarrow h$;
    $j--$;

La complexité en temps de l'algorithme 1-transitif est en $\mathcal{O}(nlogn + m)$, $m$ étant le nombre d'arcs de $G''$. C'est exactement la même que celle de 0-transitif.

## 3.3 Amélioration du modèle

### 3.3.1 Auto-copie et auto-copie inversée et complémenté

Nous avons vu qu'en biologie il pouvait survenir différents types de réarrangements génomiques. Le modèle de base les interprète par des copies et des copies inversées. Mais il existe également d'autres types de réarrangements que les copies ne peuvent expliquer. Ces réarrangements sont dûs à des duplications au sein d'une même séquence, on appelle ces duplications des auto-copies. Les duplications, comme les copies, peuvent être inversées et complémenté. Deux exemples d'auto-copies sont disponibles sur la figure 12 et 13. L'ajout de ce type d'opérations est donc intéressant car il nous permet de mieux expliquer les différences entre génomes. Il faut pouvoir détecter ce

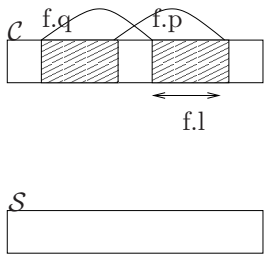


FIG. 12 – Auto-copie

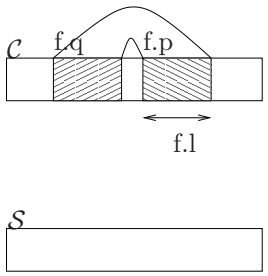


FIG. 13 – Auto-copie inversée et compémenté

genre de réarrangements. Pour cela nous utilisons Leung ou Vmatch. En effet, ces algorithmes sont

faits pour trouver des séquences communes entre deux séquences. Ainsi si on donne en entrée deux instances d'une même séquence, la sortie correspondra aux auto-copies et aux auto-copies inversées.

Il faut également définir le coût des auto-copies, une auto-copie a les mêmes paramètres qu'une copie *classique*, son codage sera donc le même (même chose pour une auto-copie inversée). Néanmoins, il faut rajouter un bit pour encoder le type de l'opération (Rappel : le coût d'une opération correspond aux nombres de bits nécessaires pour l'encoder, un entête spécifiant le type d'opération est requis) il y a maintenant 5 opérations il faut donc $log_2(5)$ bits pour encoder l'entête.

Il reste à intégrer les opérations dans le script-graph, étant donné que nous construisons la séquence cible de gauche à droite, pour utiliser une auto-copie il faut que la partie à copier ait déjà été construite. Il faut donc ajouter une contrainte sur les auto-copies telle que $f.p > f.q$. Il faut donc faire un pré-traitement pour enlever les auto-copies non conformes.

L'auto-copie pouvant être assimilée à une copie *classique*, la relation d'ordre entre tous les appariements ne change pas. Ainsi, tous les algorithmes préalablement définis fonctionnent parfaitement avec ces nouvelles opérations.

### 3.3.2 Copie-inversée

Les copies-inversées n'étant pas implémentées dans notre version de Leung, le passage à l'algorithme de Vmatch nous a permis de les intégrer dans le script-graph. Ce type d'opération pouvant être vu comme de simples copies, leurs intégrations n'ont pas posé de problème majeur.

### 3.3.3 Chevauchement

Une des caractéristiques de nos algorithmes est qu'ils ne prennent pas en compte les appariements chevauchants dans la séquence cible. Ainsi lorsqu'un chevauchement existe l'un des deux appariements est exclu alors que leur combinaison pourrait diminuer la distance. Ainsi, la gestion des chevauchements dans nos algorithmes permettrait un affinement de notre distance.

**Méthode simple** La méthode la plus simple à mettre en place est de simplement permettre les chevauchements en modifiant la définition d'ordre sur les appariements. la définition d'ordre devient donc :

**Définition 12** *Soient deux appariements $f$ et $g$, $f$ précède $g$, noté $f <_0 g$ ssi $f.p < g.p$ et $f.p + f.l < g.p + g.l$.*

Si on reprend l'exemple figure 8. On rajoute les arcs permettant d'autoriser les chevauchements via la relation d'ordre $<_0$ (avec un poids de 0), cela donne le script-graph figure 14. Le meilleur chemin dans ce graphe est le chemin en gras son poids est de 75 ce qui est meilleur que le chemin trouvé sans chevauchement. L'inconvénient de cette méthode c'est que l'on crée deux fois la partie qui

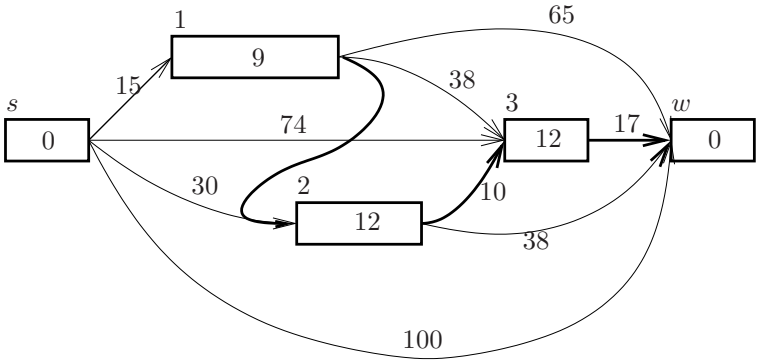


FIG. 14 – script-graph avec la relation d'ordre $<_0$ le chemin en gras est le chemin de poids minimal entre $s$ et $p$

chevauche. En effet le codage de nos opérations ne permet pas de savoir si l'on chevauche, ceci est

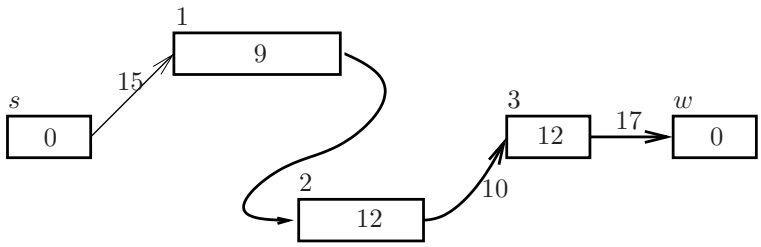


FIG. 15 – graphe $G''$ avec la relation d'ordre $<_0$ le chemin en gras est le chemin de poids minimal entre $s$ et $p$

dû au fait que dans le codage d'une opération nous ne disposons pas du début de l'appariement (on code uniquement l'offset entre $p$ et $q$, et la longueur $l$). Ainsi il faut modifier le codage des opérations en ajoutant le début de l'appariement. Pour cela nous codons le décalage entre la fin de l'opération précédente et le début de l'appariement actuel.

**Méthode de découpe** Il est clair que l'on peut modifier deux appariements chevauchants afin qu'ils ne chevauchent plus sans perdre le recouvrement initial sur la séquence cible. Ceci est illustré figure 16. Soient $f(p, q, l)$ et $g(p', q', l')$ deux appariements chevauchants tels que $f <_0 g$ si on modifie $p'$ par $p + l$ alors il n'y a plus de chevauchement et $f < g$. Il faut également modifier $q'$ selon le type de réarrangement (inversée ou non), et également mettre à jour $l'$.

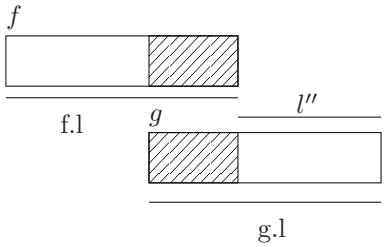


FIG. 16 – Soit $f(p, q, l)$ et $g(p', q', l')$ deux appariements chevauchants tels que $f <_0 g$

L'ajout des chevauchements est facilement intégrable dans les algorithmes précédents. Néanmoins étant donné que la relation d'ordre sur les appariements est modifiée les graphes $G'$ et $G''$ ne posséderont pas les mêmes arcs. Ainsi, l'ajout des chevauchements ne garantie pas que les heuristiques nous donnent de meilleurs chemins.

## 3.4 Implémentation

Afin de valider nos heuristiques expérimentalement nous les avons implémentées en $c++$. Pour pouvoir nous comparer aux algorithmes existants mais aussi afin de comprendre les algorithmes implémentés par J.S. Varré nous avons re-codé l'algorithme exact et valid-script. Nous avons également développé une passerelle pour pouvoir utiliser les résultats de Vmatch.

**Visualisateur** Les résultats ne sont pas simples à visualiser quand ils sont sous forme textuelle. Une méthode de visualisation classique est le *Dotplot* : une séquence est mise en abscisse et l'autre en ordonnée, un point est placé en $(x, y)$ quand la nucléotide $x$ à été mise en relation avec la nucléotide $y$. Cette visualisation peut être facilement obtenu avec gnuplot [1] mais la visualisation est trop statique : elle ne permet pas de zoom dynamique, ni de retrouver facilement les informations sur un appariement.

C'est pour pallier ce manque que nous avons décidé de développer en collaboration avec Vanessa Douttez qui est également en stage dans l'équipe MAB, un outil permettant une visualisation plus

souple. L'outil est codé en actionScript 2.0 (technologie Adobe Flash ®). Il permet donc de faire des Dotplots avec zoom figure 17, et également une autre visualisation ou les séquences sont mises en parallèle, et les appariements sont représentés par des segments liant leurs débuts dans $\mathcal{S}$ à leurs débuts dans $\mathcal{C}$ figure 18.

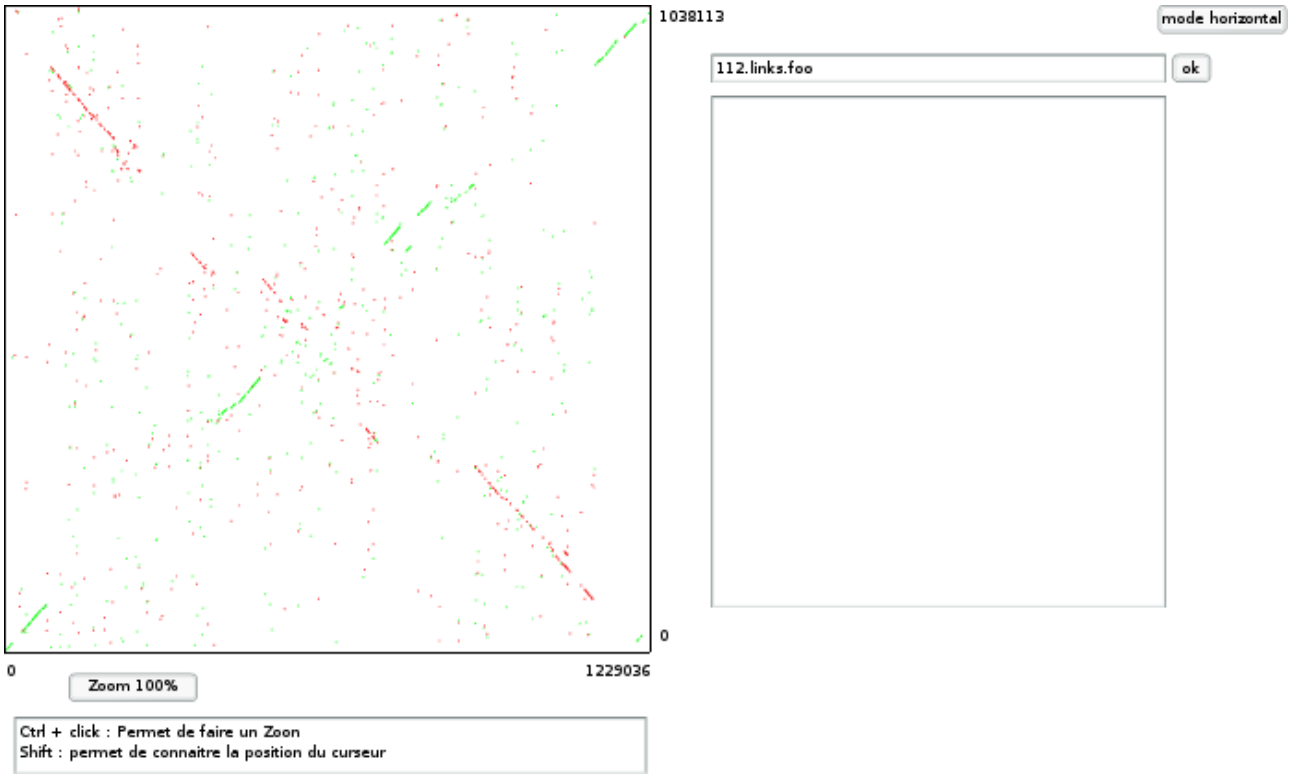


FIG. 17 – Impression d'écran du visualisateur de résultats. Cette vue montre le mode *Dotplot*. Les points en vert représentent les copies et les rouges les copies-inversées

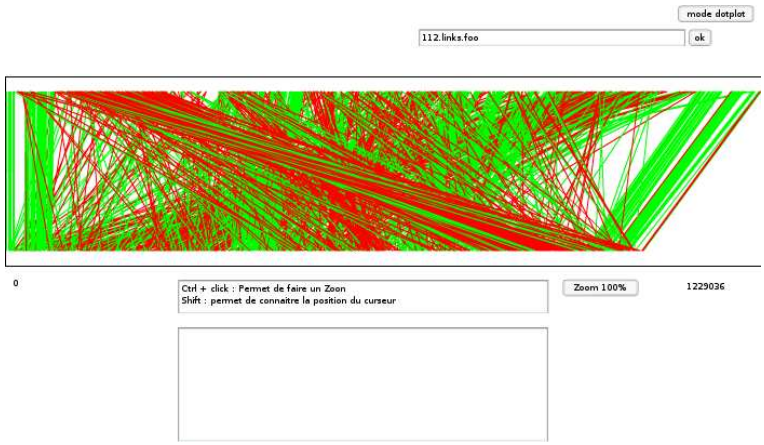


FIG. 18 – Impression d'écran du visualisateur de résultats. Cette vue montre le mode *horizontal*. Les segments en vert représentent les copies et les rouges les copies-inversées

# 4 Expérimentation des différents algorithmes

## 4.1 Tests de validation

Dans cette section nous allons voir des comparatifs sur les différents algorithmes. Nous allons tout d'abord nous intéresser aux temps d'exécutions des algorithmes sur des données aléatoires. Nous définissons $\alpha$ comme le paramètre de longueur minimale des algorithmes de recherche d'appariements, et par $\beta$ le paramètre de bruit du générateur aléatoire.

**Génération aléatoire** On commence par générer la séquence cible aléatoirement puis on altère celle-ci par des substitutions de nucléotides pour former la séquence source. Le paramètre $\beta$ permet de régler le taux de substitution.

**Exact** Le test figure 19 met en évidence que les heuristiques sont beaucoup plus rapides que l'algorithme exact malgré le fait que leurs complexités sont proches.

| | Méthodes | | | |
|---|---|---|---|---|
| taille séquences | Exact | Valid-script | 0-Transitif | 1-Transitif |
| 20500 | 50.72 | 0.79 | 0.08 | 0.12 |
| 21000 | 56.06 | 1.06 | 0.08 | 0.13 |
| ... | ... | ... | ... | ... |
| 26500 | 132.64 | 3.72 | 0.14 | 0.24 |
| 27000 | 145.46 | 7.34 | 0.15 | 0.25 |
| 27500 | 153.89 | 6.66 | 0.15 | 0.26 |
| 28000 | 170.73 | 5.78 | 0.17 | 0.28 |
| 28500 | 180.18 | 6.73 | 0.17 | 0.29 |
| 29000 | 194.55 | 6.42 | 0.18 | 0.30 |
| 29500 | 209.17 | 8.11 | 0.19 | 0.33 |

FIG. 19 – Tableau comparatif du temps d'exécution des différents algorithmes. $\alpha = 8$, $\beta = 0,10$

**Heuristiques** La figure 20 compare les temps d'exécution des heuristiques. Il est clair que ce test montre que valid-script est moins performant que 0-transitif et 1-transitif. La figure 21 montre le défaut de valid-script, plus le paramètre $\alpha$ est bas plus le nombre d'appariement est très grand et donc on a statistiquement beaucoup de chevauchements ce qui fait que le nombre d'ensembles chevauchants tend vers 1. Ainsi l'heuristique se comporte comme l'algorithme exact. Pour ce qui

est de 0-transitif et 1-transitif, 1-transitif est logiquement plus lent que 0-transitif car il évalue deux fois plus d'arcs.

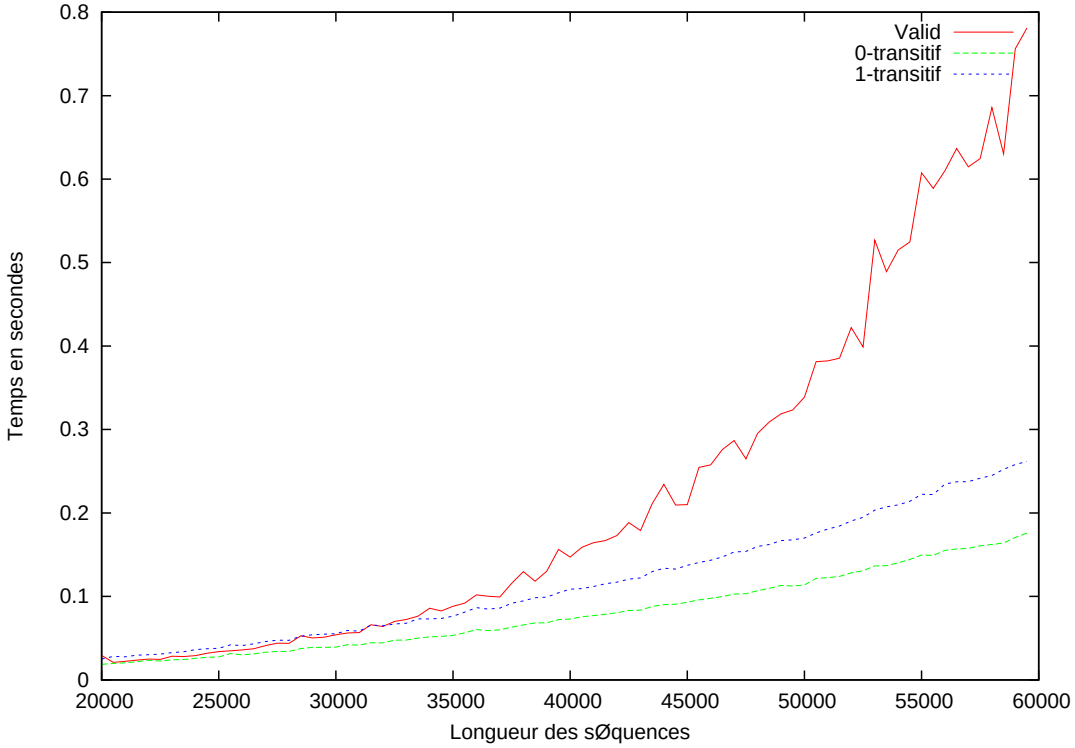


FIG. 20 – $\alpha = 9$, $\beta = 0,1$

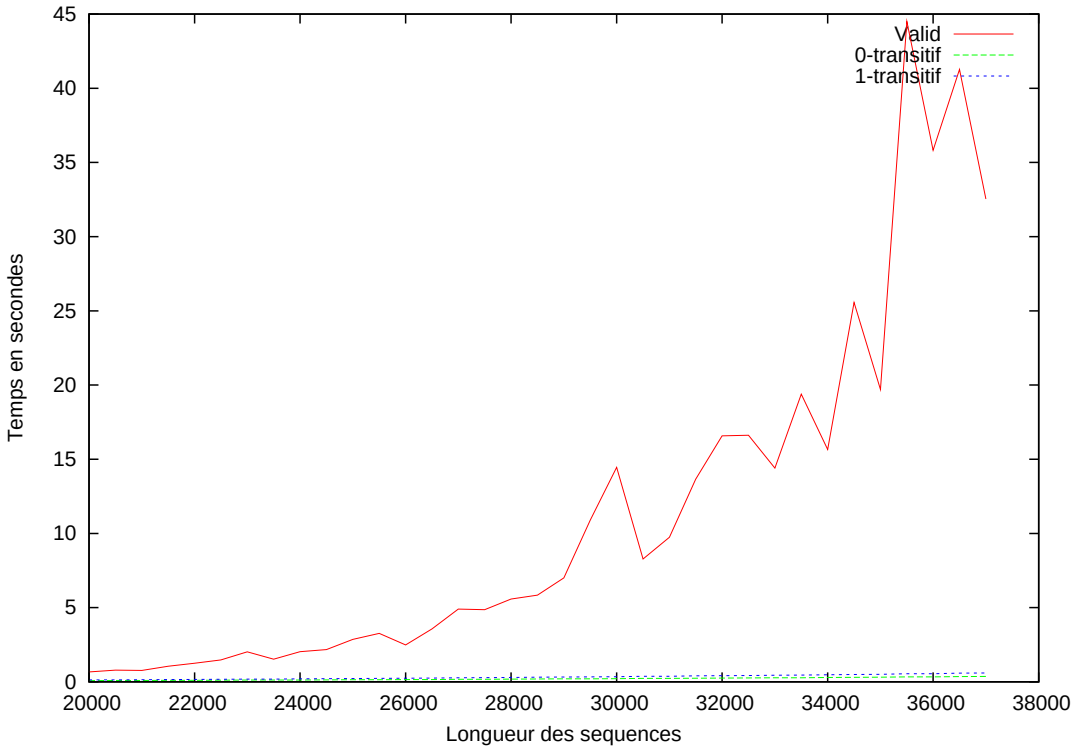


FIG. 21 – $\alpha = 8$, $\beta = 0,1$

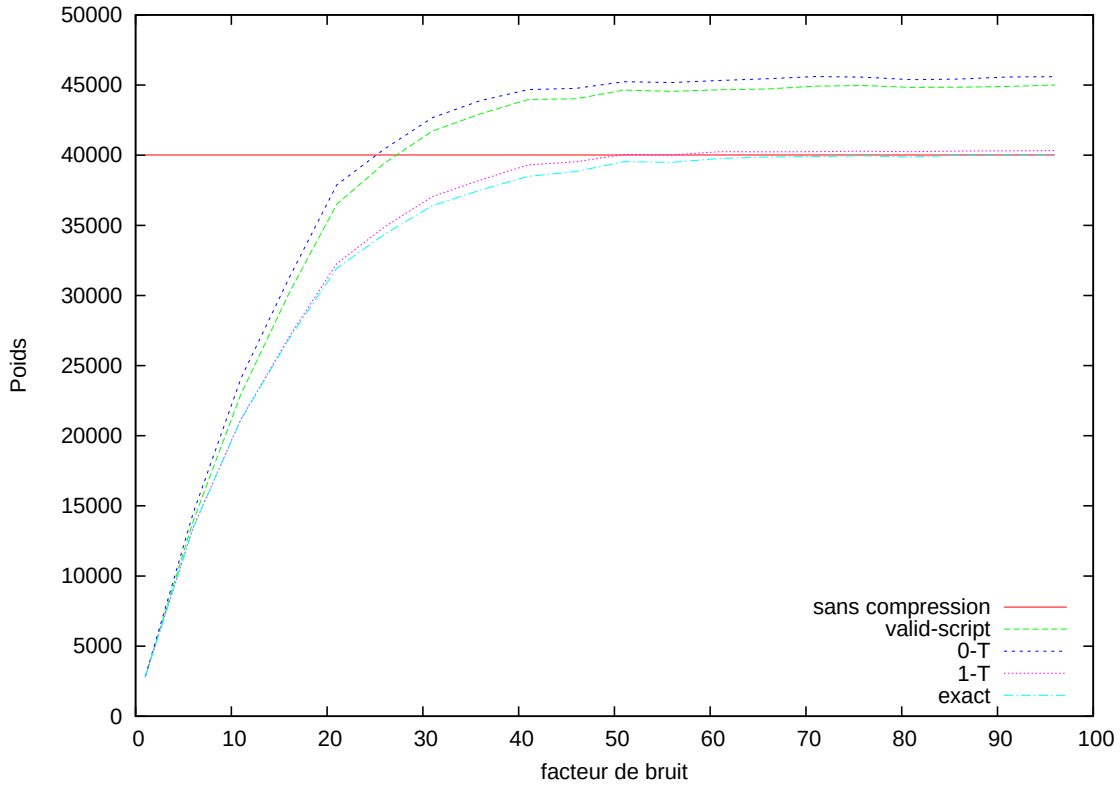


FIG. 22 – Longueur minimale appariement 9, longueur des séquences $20k$

**Aspect qualitatif des heuristiques** Le test figure 22 consiste à comparer les distances trouvées par les heuristiques en variant le paramètre $\alpha$. Ainsi nous espérons avoir un ensemble assez représentatif des distances possibles. Les résultats de ce test montrent que valid-script et 0-transitif trouvent un résultat relativement similaire, mais qu'il reste assez éloigné du résultat exact. 1-transitif est lui très proche du résultat exact. Le résulat dit sans compréssion est le poids d'un script qui ne fait qu'une insertion ainsi il ne compresse aucune donnée.

**Améliorations du modèle** L'ajout des chevauchements et des auto-copies améliore que trop peu les résultats sur les séquences aléatoires. Ceci est dû au mode de génération de ces séquences qui ne permet pas de prendre en compte ce type de mécanisme. Ces améliorations seront testées sur des séquences réelles dans la section suivante.

**Conclusion** Un des premiers bilans que nous pouvons tirer de ces tests, c'est que les heuristiques développées sont meilleures en temps que valid-script. Au niveau qualitatif 0-transitif trouve des résultats proches de ceux de valid-script, 1-transitif est lui proche du résultat exact.

## 4.2 Application à des génomes bactériens

### 4.2.1 Génomes sélectionnés

Cette section présente les tests effectués sur des données réelles. Nous avons sélectionné 12 génomes complets de différentes espèces. Le tableau ci-après en fait la liste.

| numéro | nom du génome | taille |
|---|---|---|
| 1 | Chlamydia trachomatis 434/Bu | 1,038,842 |
| 2 | Buchnera aphidicola str. Bp (Baizongia pistaciae) | 615,980 |
| 3 | Shigella flexneri 2a str. 301 | 4,607,203 |
| 4 | Escherichia coli O157 :H7 EDL933 | 5,528,445 |
| 5 | Chlamydia trachomatis D/UW-3/CX | 1,042,519 |
| 6 | Staphylococcus aureus subsp. aureus COL | 2,809,422 |
| 7 | Staphylococcus aureus subsp. aureus MW2 | 2,820,462 |
| 8 | Staphylococcus aureus subsp. aureus N315 | 2,814,816 |
| 9 | Chlamydia trachomatis L2b/UCH-1/proctitis | 1,038,869 |
| 10 | Chlamydophila pneumoniae CWL029 | 1,230,230 |
| 11 | Buchnera aphidicola str. Sg (Schizaphis graminum) | 641,454 |
| 12 | Chlamydophila pneumoniae AR39 | 1,229,853 |

FIG. 23 – Tableau des génomes sélectionnés pour les tests

### 4.2.2 Temps d'exécutions des différents algorithmes

Voici un comparatif du temps d'exécution des heuristiques sur un ensemble de comparaisons des génomes sélectionnés, l'algorithme exact étant trop lent il n'a pas été exécuté dans les tests. Le temps est en seconde.

| $\mathcal{C}$ | $\mathcal{S}$ | 0-transitif | 1-transitif | valid-script |
|---|---|---|---|---|
| 4 | 6 | 28.96 | 47.70 | 233.42 |
| 4 | 7 | 27.17 | 47.46 | 233.35 |
| 4 | 8 | 26.98 | 47.82 | 233.19 |
| 4 | 9 | 7.28 | 12.53 | 34.13 |
| 4 | 10 | 7.43 | 13.18 | 75.80 |
| 4 | 11 | 9.76 | 17.57 | 119.37 |
| 4 | 12 | 8.31 | 14.29 | 37.73 |
| 5 | 1 | 1.91 | 3.01 | 10.76 |
| 5 | 2 | 1.56 | 2.52 | 3.15 |
| 5 | 3 | 3.14 | 4.50 | 3.23 |
| 5 | 4 | 3.94 | 5.65 | 4.19 |
| 5 | 6 | 4.31 | 6.54 | 6.11 |
| 5 | 7 | 4.36 | 6.56 | 6.11 |
| 5 | 8 | 4.26 | 6.49 | 6.02 |
| 5 | 9 | 1.90 | 2.99 | 10.71 |
| 5 | 10 | 1.59 | 2.56 | 3.15 |
| 5 | 11 | 2.29 | 3.82 | 6.12 |
| 5 | 12 | 2.17 | 3.35 | 3.26 |
| 6 | 1 | 7.69 | 13.36 | 20.32 |
| 6 | 2 | 38.28 | 72.42 | 883.81 |

FIG. 24 – Tableau comparatif des temps d'exécutions des heuristiques

Si l'on compare 0-transitif à 1-transitif il est clair que l'on retrouve des résultats cohérents avec la théorie. En effet, les temps de 1-transitif sont environ deux fois plus lents que 0-transitif, pour rappel 1-transitif évalue deux fois plus d'arcs que 1-transitif ainsi ce résultat est normal.

0-transitif est clairement plus rapide que valid-script ce qui est également un résultat logique. Les performances de valid-script sont très fluctuantes : parfois très lentes et d'autre fois plus rapides que 1-transitif. Mais ceci est lié aux chevauchements : quand il y a beaucoup de chevauchements, beaucoup d'arcs sont ajoutés et l'algorithme est lent, et quand il n'y a peu de chevauchement valid-script à des performances égales à 1-transitif voire meilleures car il a logiquement moins d'arcs.

### 4.2.3 Aspect qualitatif des heuristiques

Maintenant nous allons nous intéresser aux résultats des heuristiques. Les résultats indiqués sont les pourcentages de compression que donnent les algorithmes.

| $\mathcal{C}$ | $\mathcal{S}$ | 0-transitif | 1-transitif | valid-script |
|---|---|---|---|---|
| 1 | 5 | 6,6 | 6,38 | 6,48 |
| 1 | 6 | 105,23 | 100,42 | 104,71 |
| 1 | 7 | 105,22 | 100,43 | 104,71 |
| 1 | 8 | 105,25 | 100,43 | 104,73 |
| 1 | 9 | 6,66 | 6,44 | 6,48 |
| 3 | 2 | 104,83 | 99,67 | 102,09 |
| 3 | 4 | 41,19 | 39,52 | 39,81 |
| 3 | 5 | 106,37 | 100,2 | 104,17 |
| 4 | 2 | 103,79 | 99,45 | 100,63 |
| 4 | 3 | 25,6 | 24,43 | 24,5 |
| 4 | 5 | 106,13 | 100,15 | 103,27 |

FIG. 25 – Tableau comparatif des résultats des heuristiques

Ces résultats sont cohérents par rapport aux test avec les séquences aléatoires, à savoir que 1-transitif trouve de meilleurs résultats que les deux autres heuristiques. Il est à noter que les différences restent assez faibles, ce qui fait de 0-transitif un algorithme assez intéressant.

### 4.2.4 Améliorations du modèle

Maintenant nous allons nous intéresser à l'ajout de la gestion des chevauchements (avec la méthode de découpe) et l'ajout des auto-copies. Pour cela nous prenons l'algorithme 1-transitif pour référence.

| $\mathcal{C}$ | $\mathcal{S}$ | 1-transitif | auto-copie | chevauchement | auto-copie+chevauchement |
|---|---|---|---|---|---|
| 1 | 5 | 6,48 | 6,37 | 5,5 | 5,48 |
| 1 | 6 | 104,71 | 97,54 | 100,43 | 97,32 |
| 1 | 7 | 104,71 | 97,8 | 100,42 | 97,58 |
| 1 | 8 | 104,73 | 96,84 | 100,44 | 96,63 |
| 1 | 9 | 6,48 | 6,44 | 6,44 | 6,39 |
| 3 | 2 | 102,09 | 98,22 | 99,68 | 97,85 |
| 3 | 4 | 39,81 | 34,6 | 39,07 | 33,95 |
| 3 | 5 | 104,17 | 99,61 | 100,29 | 99,67 |
| 4 | 2 | 100,63 | 98,02 | 99,35 | 97,57 |
| 4 | 3 | 24,5 | 21,28 | 23,74 | 20,61 |
| 4 | 5 | 103,27 | 99,58 | 100,24 | 99,62 |

FIG. 26 – Tableau comparatif des résultats de 1-transitif, avec l'ajout des auto-copies, des chevauchements et de la combinaison des deux.

Il est clair sur ce tableau que l'ajout des chevauchements et des auto-copies réduit le taux de compression.

# 5 Conclusion et perspectives

## 5.1 Conclusion

Durant ce stage nous avons étudié le problème de mesure de ressemblance entre deux objets appliqué aux séquences génétiques. Pour cela nous avons étudié la *distance de transformation* introduite par l'article [7]. Le problème ne pouvant pas être résolu de façon optimale sur des données réelles, seule des approximations sont possibles. Nous avons créé de nouvelles heuristiques qui d'après les expérimentations donnent de meilleurs résultats que celles précédemment développées. Nous avons également amélioré le modèle initial en ajoutant le concept d'auto-copie, qui permet de mieux expliquer le passage entre deux séquences. Ainsi que l'introduction des chevauchements qui permet un affinement de la distance.

## 5.2 Perspectives

Il reste encore des voies à explorer qui permettraient d'améliorer le travail effectué. Les appariements approximés qui permettraient une meilleure compression, leur gestion modifie le calcul du poids des appariements car il faut coder les différences entre les deux fragments. Améliorer la gestion des chevauchements : nous avons fait un découpage assez simple des appariements chevauchants, une méthode plus complexe pourrait donner de meilleurs résultats. Un pré-traitement sur les appariements pourrait également permettre d'en diminuer le nombre, reste à définir les critères pertinents.

# Références


[1] http ://www.gnuplot.info/.

[2] http ://www.vmatch.de/.

[3] H Chiapello, I Bourgait, F Sourivong, G Heuclin, A Gendrault-Jacquemard, M-A Petit, and M El Karoui. Systematic determination of the mosaic structure of bacterial genomes : species backbone versus strain-specific loops. *BMC Bioinformatics*, 6(1) :171, 2005.

[4] Ming-Ying Leung, Blaisdell B.E., Burge C., and Karlin S. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *J. Mol. Biol.*, 1991.

[5] Ming Li and Paul M.B. Vitanyi. *Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2nd edition, 1997.

[6] J.-S. Varré. *Concepts et algorithmes pour la comparaison de séquences génétiques : une approche informationnelle*. PhD thesis, July 2000.

[7] Jean-Stephane Varré, Jean-Paul Delahaye, and Eric Rivals. Transformation distances : a family of dissimilarity measures based on movements of segments. *Bioinformatics*, 1999.