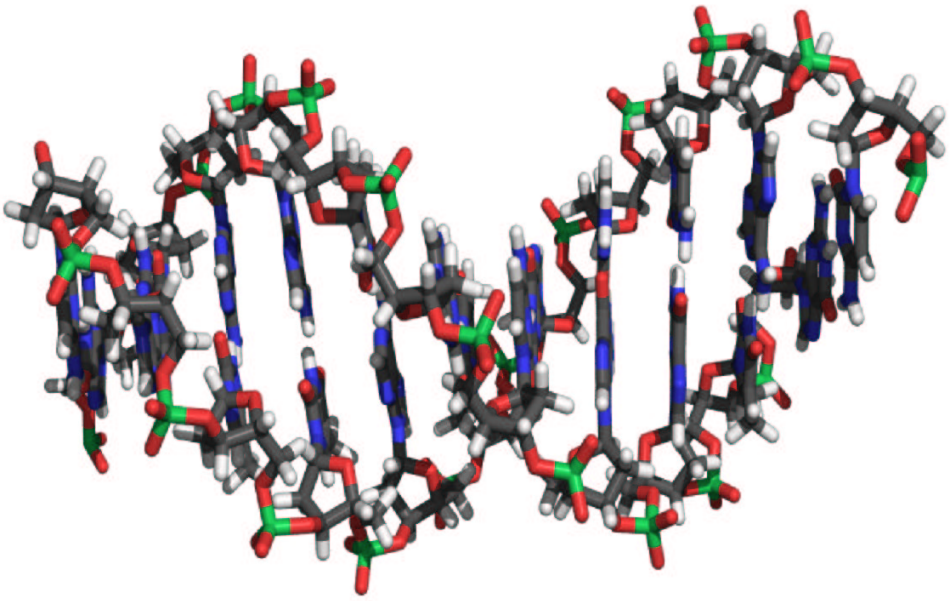
# RECHERCHE DE RÉPÉTITIONS DISTANTES DANS LES SÉQUENCES



Mardi 2 Juillet 2002


stagiaire : Laurent NOE

encadrant : Gregory KUCHEROV

jury : Alexander BOCKMAYR
Noëlle CARBONELL
Didier GALMICHE
Dieter KRATSCH
Dominique MERY

---

Université HENRI POINCARÉ
Ecole Doctorale IAEM-LORRAINE / DEA Informatique de Lorraine
LORIA / Equipe ADAGE


---

**Résumé**

Les répétitions apparaissent de manière très importante dans le génome humain ; on peut considérer qu'elles entrent dans plus de la moitié de sa composition. Ces répétitions ont une particularité : leurs occurrences ne sont pas des copies exactes mais approchées ; un certain nombre d'erreurs modifient partiellement au moins l'une des copies de la répétition.

Afin de pouvoir détecter de telles répétitions, l'idée étudiée dans le cadre de ce stage se fonde sur une constatation qui est la suivante : des répétitions *approchées* de taille conséquente contiennent des sous fragments répétés de manière *exacte*. Notre démarche diffère de certaines approches existantes en ce sens qu'elle considère, non pas un, mais des regroupements de sous fragments répétés de manière exacte. Elle s'appuie pour cela sur des propriétés statistiques des séquences à rechercher.

L'algorithme présenté a été implanté et des tests comparatifs ont été réalisés vis à vis des logiciels `ASSIRC` et `BLAST`.

**Remerciements**

*Je voudrais remercier les personnes qui ont, directement ou indirectement, contribué au bon déroulement de mon stage au sein du Loria. En particulier,*

*merci à mon encadrant Grégory Kucherov pour son aide et sa disponibilité tout au long de ce stage*

*à Marie-Pierre Etienne et Pierre Valois, toute ma reconnaissance pour leurs conseils et tout le travail qu'ils auront réalisé afin de comprendre et de résoudre nos problèmes liés aux probabilités*

*mes remerciement à Gilles Schaeffer pour ses renseignements sur les marches aléatoires et à Ghizlane Bana pour ses explications au sujet des aspects biologiques des séquences d'ADN*

# Table des matières

# 1 Introduction

Le génome constitue le seul support durable biochimique de l'hérédité. Il contient l'information nécessaire pour construire et maintenir un organisme vivant. A quelques exceptions près, il se compose d'un ensemble de molécules d'ADN[1] qui forme les chromosomes (seuls certains virus ont un génome composé d'une molécule analogue nommée ARN[2]).

La molécule d'ADN est construite à partir d'un double chaînage de nucléotides prises parmi quatre bases A,T,G,C (Adénine, Thymine, Guanine, Cytosine).

Depuis la reconnaissance par la communauté scientifique de cette molécule comme support de l'hérédité (années 50), des techniques ont été mises au point afin d'extraire l'information qu'elle contenait. Ces techniques dites de séquençage ont permis d'isoler des fragments de chromosomes, puis des chromosomes entiers (plusieurs centaines de milliers de nucléotides au minimum).

L'analyse de ces séquences a permis par la suite d'isoler un certain nombre de gènes, auxquels des fonctions ont quelquefois pu être attribuées. Cette analyse est désormais menée de manière semi-automatique. Un des aspects automatisables intéressants concerne la recherche de répétitions locales dans des séquences d'ADN.

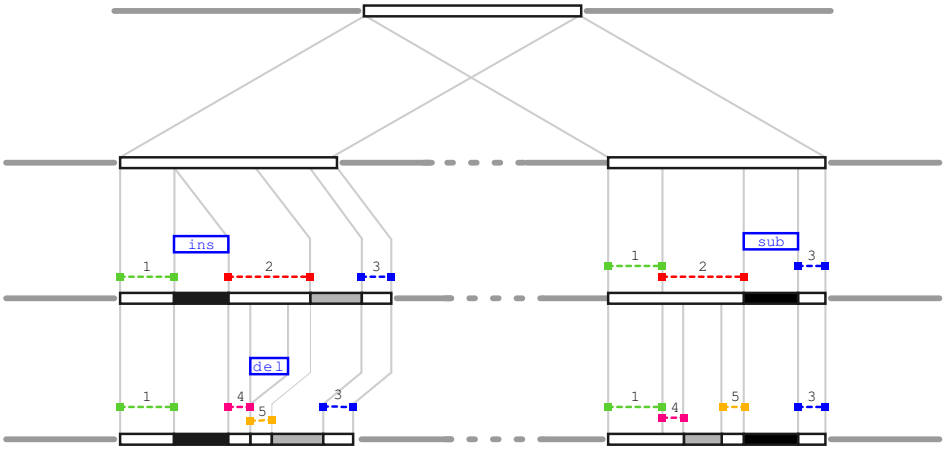



FIG. 1 – *Evolution des copies d'une répétition : une duplication est réalisée lors de la première étape. Les copies évoluent ensuite séparément durant les deux étapes suivantes : une insersion d'éléments (ins) sur la copie de gauche, une substitution d'éléments (sub) sur celle de droite, puis une suppression d'éléments (del) remanient les répétitions exactes.*

C'est sur ce problème de recherche de répétitions que ce stage trouve son origine : il s'agit ici de mettre en place un algorithme de recherche de répétitions spécifique à l'ADN.

[1] Acide Désoxyribonucléique
[2] Acide Ribonucléique

Les problèmes de recherche de répétitions sont bien connus dans le domaine de l'algorithmique du texte, et des algorithmes efficaces existent dans le cas de répétitions exactes. Malheureusement, ils ne sont pas applicables (du moins, pas directement) à l'ADN, car les répétitions se sont transformées durant leur évolution (par insertions, suppressions ou substitutions de nucléotides sur les copies[3]). Celles ci sont devenues des répétitions dites *approchées*.

Cependant, malgré cette évolution des copies, des sous fragments répétés de manière exacte subsistent dans une répétition approchée. Afin de pouvoir détecter une répétition approchée, une méthode classique consiste à rechercher les répétitions exactes[4], et d'essayer de les étendre par des méthodes algorithmiques afin de trouver des répétitions approchées.

Notre approche se base sur une idée similaire mais essaye, à l'aide de critères statistiques dans un premier temps, de déterminer la distribution de ces sous fragments répétés exacts, puis dans un deuxième temps, d'en rassembler un certain nombre afin de délimiter des zones pouvant contenir une répétition approchée. Il est nécessaire pour cela de prendre en compte les propriétés statistiques des répétitions recherchées afin de rassembler les sous fragments répétés exacts.

# 2 Etat de l'art

La recherche de similitudes dans des séquences est un problème complexe qui ne peut être mené aujourd'hui de manière exacte à l'aide d'algorithmes combinatoires que sous certaines hypothèses assez strictes. Des algorithmes heuristiques ont donc été créés de manière à trouver un maximum de répétitions en un temps raisonnable.

La diversité des approches algorithmiques est importante mais des principes et des sous méthodes sont communs à certains algorithmes (comme, par exemple, l'utilisation de la liste chaînée des $k$-mots, les méthodes dites d'extension des graines, etc.)

Les développements effectués offrent un certain nombre de logiciels destinés à la recherche de répétitions. Certains logiciels d'alignement comme `BLAST` utilisent des résultats récents issus des probabilités afin de déterminer des similarités « significatives » entre séquences (elles ont peu de chance d'être simplement dues au hasard). D'autres logiciels comme `Tandem Repeats Finder` ou `mreps` se sont spécialisés sur un type de répétition particulier (en l'occurrence, les répétitions en *tandem*) en utilisant chacun une approche toute particulière.

[3] on parle alors de « mutations »

[4] ces répétitions sont souvent dénommés « graines »

## 2.1 Répétitions Quelconques

### 2.1.1 BLAST

BLAST (et ses nombreuses formes dérivées) est certainement le logiciel le plus utilisé pour l'analyse de séquences dans le monde de la bioinformatique. Il implante un algorithme heuristique très rapide. Son rôle est de rechercher des zones similaires entre une base de données d'ADN et une séquence à traiter. On l'utilise également pour la recherche de similitudes entre deux séquences.

BLAST est un logiciel d'alignement de séquences : un alignement est ici vu comme une comparaison de deux sous-fragments de séquences, en n'autorisant, ni les insertions, ni les suppressions (modèle initial de BLAST). Cette comparaison élément par élément (lettre à lettre) amène à effectuer uniquement des substitutions d'éléments lors de l'alignement.

Dans le cas d'acides aminés, une matrice de substitution $s$ évalue pour cela un score pour chaque substitution en fonction des deux éléments à aligner (Ce score est très élevé si les éléments sont les mêmes, et décroît en fonction du taux de ressemblance des propriétés chimiques des éléments).

Le score local maximal $H_n$ sur un alignement donné de taille $n$ est défini comme

$$H_n = \max_{1 \leq i \leq j \leq n} \sum_{k=i}^{j} s_k \tag{1}$$

où $s_k$ est le score unitaire calculé pour deux éléments à aligner situés à la position $k$.

Dans le cas aléatoire, ce score peut être évalué sur un alignement de deux séquences dont les probabilités des éléments sont données. Le $i^{me}$ élément de l'alphabet des bases a une probabilité d'apparition $p_i$ dans la première séquence (respectivement $p'_i$ dans la deuxième).

En émettant deux hypothèses sur le score,

– au moins un score unitaire $s_{ij}$ est positif,

– $E = \sum_{i,j} p_i p'_j s_{ij} < 0$,

KARLIN [KA90] déduit que la variable aléatoire $H_n$ peut être approximée par

$$\lim_{n \to \infty} \mathcal{P}\Big\{ H_n > \frac{\ln n}{\lambda^*} + x \Big\} = \exp(-K^* e^{-\lambda^* x}) \tag{2}$$

(Voir également [Mer99, Nic97]) Les constantes $K^*$ et $\lambda^*$ sont évalués respectivement par un encadrement et une résolution d'équation.

Cette approximation dans le cas aléatoire permet de distinguer la « significativité » de scores locaux dans le cas de séquences réelles, et donc de savoir si les similitudes trouvées ont peu de chance d'être dues au hasard.

### 2.1.2 Reputer

Reputer [KS99, KOSS01] était à l'origine un logiciel de recherche de répétitions exactes basé sur la construction de l'arbre des suffixes (il n'autorisait pas

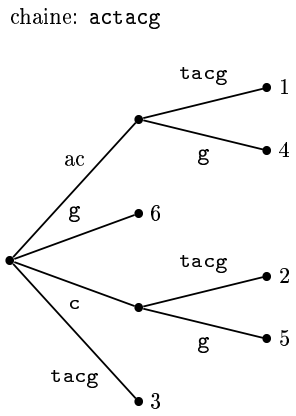



FIG. 2 – Un exemple d'arbre des suffixes

d'erreurs entre les copies). L'arbre des suffixes permet d'obtenir l'ensemble des répétitions du texte et se construit en temps et en espace linéaire.

Un deuxième module a été ajouté afin de compléter la recherche et pouvoir considérer les répétitions approchées. Ce module se base sur les « graines » (répétitions exactes) trouvées à l'aide du premier module. Il essaie d'étendre ces graines selon un certain nombre de critères pour détecter des répétitions approchées de taille supérieure aux graines.

L'extension utilise deux algorithmes :

- Le premier (`MMR`) tente d'étendre les graines sur les extrémités afin de découvrir éventuellement une répétition ayant $k$ erreurs selon la distance de Hamming. Il n'autorise donc pas les insertions et les suppressions d'éléments mais uniquement les substitutions.
- Le deuxième (`MDR`) a aussi pour but de trouver des extensions à certaines graines mais en autorisant les insertions et les suppressions d'éléments : il utilise donc la distance d'édition, et se base sur un algorithme de programmation dynamique effectué sur un domaine limité.

### 2.1.3 Oracle des Facteurs

L'oracle des facteurs [ACR99] est une structure de données permettant de représenter tous les facteurs d'un mot. Pour un mot $M$ de taille $n$, un automate est créé, et possède les caractéristiques suivantes :

- il possède exactement $n + 1$ états qui sont tous terminaux
- son nombre de transitions est compris entre $n$ et $2n - 1$ et se divise en deux classes :
  - des transitions internes, toujours au nombre de $n$, permettant de passer

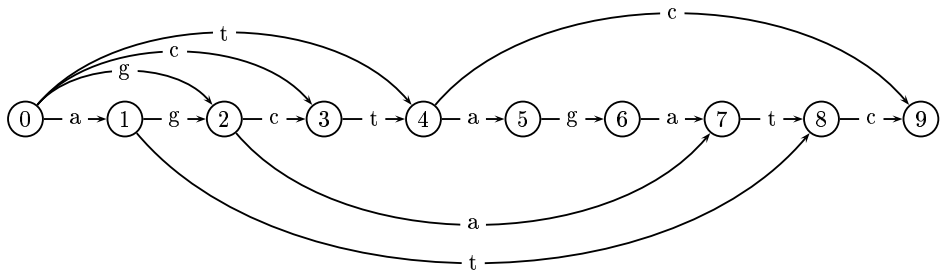



FIG. 3 – Oracle des facteurs de la chaîne `agctagatc`

d'un préfixe de longueur $i$ à un préfixe de longueur $i+1$

– des transitions externes, permettant de passer d'un préfixe de longueur $i$ à un préfixe de longueur $i+j$ ($j > 1$)

L'oracle permet de reconnaître au moins tous les facteurs de $M$. Il est utilisé pour la recherche de répétitions exactes et a aussi été étendu à la recherche de répétitions approchées afin d'être utilisé sur des séquences d'ADN.

### 2.1.4 Méthode de [BBK91]

Une autre approche proposée par KARLIN consiste à utiliser dans un premier temps la liste chaînée des $k$-mots (Expliquée au chapitre 4) pour trouver les répétitions exactes de taille maximale. Un $k$-mot est un fragment de texte de taille $k$. La liste des $k$-mots permet d'obtenir, pour n'importe quel $k$-mot donné, toutes les positions de ses occurrences sur le texte.

Aprés avoir découvert les répétitions exactes du texte, celles qui sont de taille suffisante (nommés *core blocks*) sont mémorisées selon un certain nombre de critères, comme leur nombre dans la séquence, dans un groupe de séquences, etc. Après avoir dégagé un ensemble de *core blocks* intéressants, une extension autorisant des erreurs est testée sur chacune des occurrences.

On évalue, pour chaque occurrence d'un *core block* donné, les $k$-mots juxtaposés à gauche (respectivement à droite) de cette occurrence, en supprimant, après chaque évaluation, le nucléotide frontalier au *core block*.

Cela donne, pour le texte $\overline{\texttt{ATCA}}\texttt{C}\underline{\texttt{GGGAGAGG}}$ où le *core block* est souligné, une suite de $k$-mots ($k = 3$) `CAC`,$\overline{\texttt{TCA}}$,`ATC`,...

Cette évaluation est faite pour chacune des $t$ occurrences d'un *core block* en se limitant à $\epsilon$ suppressions par occurrence, ce qui donne $\epsilon \cdot t$ $k$-mots placés dans un tableau en deux dimensions : le plus redondant de ces $k$-mots sera alors celui qui

servira d'extension.

| | |
|---|---|
| cb1 : | TCGCCCCTCTGGGGAGAGGGTTAGGGTGAG |
| cb2 : | CTCGCCCCCTCGGGGAGAGGGTTAGGGTGAG |
| cb3 : | TCTCCCCCTCGGGGAGAGGATTAGGGTGAG |

| erreur | cb1 | cb2 | cb3 |
|---|---|---|---|
| 0 | CTG | CTC | TCG |
| 1 | TCT | CCT | CTC |
| 2 | CTC | CCC | CCT |

| $k$-mot | N | positions |
|---|---|---|
| CTC | 3 | (cb2,0),(cb3,1),(cb1,2) |
| CCT | 2 | (cb2,1),(cb3,2) |

TAB. 1 – *Extensions de* core blocks *: 3 occurrences (*$cb1$,$cb2$,$cb3$*) d'un bloc* GGGAGAGG *ont été découvertes sur le texte (tableau 1). On supprime alors des nucléotides juxtaposées à gauche de chacune des occurrences . Les* $k$*-mots issus de ces extensions (tableau 2) sont comptabilisés (tableau 3).*

### 2.1.5 ASSIRC

ASSIRC [VBA+98] est également un logiciel de recherche de répétitions approchées. Il est développé par une équipe de l'ENS Paris.

Il prend en entrée la liste des $k$-mots. A partir de cette liste, il détermine les couples de répétitions exactes de taille $k$. Ces derniers serviront de « graines » à un algorithme d'extension.

L'extension est faite à l'aide d'une marche aléatoire : une marche aléatoire est une transformation injective qui est utilisée dans ce cas pour savoir si deux sous séquences sont potentiellement ressemblantes. On associe, pour chaque lettre de l'alphabet, un vecteur de déplacement dans un plan, et l'on trace pour chaque sous séquence lue, sa courbe (succession des déplacements) générée en partant de l'origine.

Deux sous séquences ressemblantes vont voir leur courbes respectives évoluer dans un espace proche. En contrepartie, il n'est absolument pas sûr que deux courbes proches soient un témoin *garanti* de deux séquences ressemblantes.

Les critères retenus par ASSIRC pour considérer qu'un couple de séquences est potentiellement une répétition (et pour éliminer les autres couples) sont basés sur la distance euclidienne entre les deux courbes évalué à chaque pas. Un seuil maximal $\delta$ est fixé sur cette distance. Si ce seuil est dépassé durant un nombre d'étapes $\tau$, alors l'extension s'arrête.

## 2.2 Répétitions en tandem

Les répétitions en tandem sont identifiables par le fait que les copies (au nombre de deux, trois, voire quelquefois plus de cent) sont juxtaposées entres elles et non distantes. Dans cette famille, des répétitions appelées *micro-satellites* sont composées de fragments courts répétés et juxtaposés des centaines de fois. Ils sont quel-

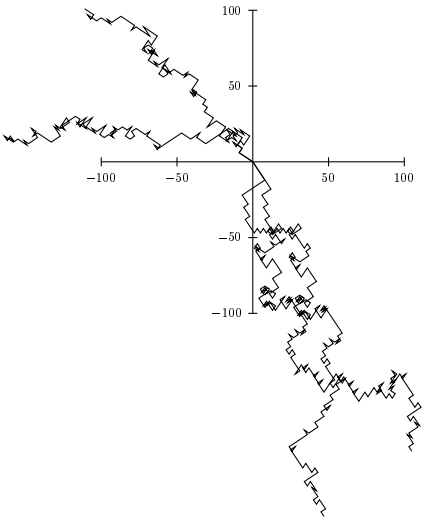



FIG. 4 – Un exemple de tracé de marche aléatoire

quefois responsables de maladies génétiques graves (comme la maladie de Huntington).

A cause du caractère particulier de ces répétitions, des algorithmes et des logiciels spécifiques ont donc été créés pour les identifier dans le génome.

### 2.2.1 `mreps`

`mreps` [KK99, GK00] est un logiciel de recherche de répétitions en tandem développé et maintenu par l'équipe ADAGE au LORIA. Il est basé sur un algorithme entièrement combinatoire et permet de trouver toutes les répétitions en tandem dont les copies sont exactes. Il peut également travailler avec des copies approchées ($k$ erreurs en considérant la distance de Hamming).

Il utilise la $s$-factorisation du texte et des structures comme le DAWG (Direct Acyclic Word Graph) pour réaliser cette tache.

### 2.2.2 `Tandem Repeats Finder`

L'algorithme proposé par BENSON [Ben98] et implémenté dans `Tandem Repeats Finder` [Ben99] permet, à partir de critères statistiques, et en utilisant au départ la liste chaînée des $k$-mots, de localiser des répétitions en tandem avec un pourcentage d'erreur maximal toléré entre les copies. Les critères sont calculés selon un certain nombre de paramètres donnés par l'utilisateur concernant les répétitions en tandem recherchées.

L'algorithme construit, à partir de la liste des $k$-mots, une liste des distances. Il s'agit en fait d'un ensemble de listes : chaque liste $D_d$ a pour rôle de conserver les couples de $k$-mots rencontrés dont la distance intra-couple était $d$. Cette liste des distances, lorsqu'elle est mise à jour (ajout d'un couple à $D_d$), est alors utilisée pour détecter une éventuelle répétition en tandem de période approchant $d$ à l'aide des critères calculés.

Certains critères utilisés par ce programme nous ont inspirés pour la réalisation de notre algorithme de recherche de répétitions approchées (présenté au chapitre 5). Une comparaison des critères utilisés par les deux algorithmes sera réalisé au chapitre 6.

# 3 Probabilités

Nous avons vu (de manière intuitive) lors de l'introduction que des répétitions approchées contenaient des sous fragments répétés de manière exacte. Nous allons détailler dans ce chapitre quelques résultats issus des probabilités qui nous permettrons de caractériser ces répétitions exactes. Pour cela, le choix de deux modèles probabilistes (*lancer de pièce* et *marche aléatoire*), et l'étude d'un certain nombre de variables aléatoires seront réalisés.

## 3.1 *Coin Tossing* (lancer de pièce)

La comparaison de deux occurrences d'une répétition approchée peut être modélisée par une suite binaire, si l'on suppose qu'il n'y a pas d'*indels* : cette suite est construite simplement en prenant le résultat booléen de la comparaison des lettres alignées.

```
ATTAGCCACGCGT
ATCGGCTACGGGT
1100110111011
```

Les variables aléatoires provenant du lancer de pièce sont alors intéressantes : on considérera qu'un *pile* sur un lancer donné équivaudra à une égalité entre lettres alignées donc sera équivalent à la valeur `1`.

### 3.1.1 *Longest Head Run* (plus long train de piles)

On recherche la plus longue répétition exacte contenue dans deux occurrences d'une répétition approchée . On suppose connaître le taux de ressemblance $p$ entre les deux occurrences ainsi que leurs taille $n$. En utilisant le modèle précédent, cela revient à chercher la plus longue série ininterrompue de `1`, donc le plus long train de piles en $n$ lancers.

On effectue $n$ lancers d'une pièce biaisée ($\mathcal{P}\{pile\} = p$). On veut déterminer le plus long train ne contenant que des piles (PLTP) dans ces $n$ lancers.

Lorsque $n$ est grand, le nombre de piles attendus en $n$ lancers est de $n(1-p)$. En négligeant les effets de bord, le nombre de trains de piles de taille $\geq 1$ est $\approx n(1-p)p$. Par récurrence, le nombre de trains de piles de taille $\geq r$ est $\approx n(1-p)p^r$. En supposant que le plus long train de piles est unique, on peut alors utiliser cette hypothèse et écrire une « pseudo-équation » $n(1-p)p^r \approx 1$ d'où l'on déduit que la taille $r$ du plus long train de piles est alors approché par [Jag01]

$$r = \log_{1/p}[n(1-p)] \tag{3}$$

De manière plus rigoureuse (mais sans démonstration à priori), le plus long train de piles peut être approché lorsque $n$ tend vers l'infini par la distribution [GSW86]

$$(W + \log(n(1-p)))/\lambda \tag{4}$$

où $\lambda = \log(1/p)$ et W est un variable aléatoire telle que sa distribution est $F(w) = \exp(-\exp(-w))$.

On considère alors une valeur $\alpha$ qui est le taux d'erreur toléré de « rater » ce plus long train de piles par valeur supérieure.

On déduit de la formule (4) que si l'on tolère de « rater », par valeur supérieure, $\alpha\%$ des plus longs trains de piles, il faudra prendre une valeur seuil $w_\alpha$ sur la distribution telle que $F(w_\alpha) = \alpha$, d'où l'on déduit :

$$w_\alpha = -\log(\log(\frac{1}{\alpha})) \tag{5}$$

On détermine alors que la taille minimale tolérée du plus long train de piles est donnée par :

$$(\log(n(1-p)) - \log(\log(1/\alpha)))/\lambda \tag{6}$$

Cette loi n'est valide que si $n$ tend vers l'infini. Afin d'évaluer son comportement pour des valeurs de $p$ et de $n$ fixées, une comparaison entre cette loi et la simulation du *PLTP* a été réalisée (elle est disponible en annexe).

**Utilisation :** d'après cette formule, et en utilisant le modèle d'alignement binaire, on peut déduire le critère suivant : on trouvera avec un taux d'erreur $\alpha$ , dans une répétition approchée de taille au moins $n$ ayant un taux de ressemblance entre copies d'au moins $p$, une répétition exacte de taille $\geq k$ donné par la formule (6).

En recherchant des répétitions exactes de cette taille, on est alors quasiment sûr d'en trouver au moins une dans les répétitions approchées considérées.

### 3.1.2 *Waiting Time* (temps d'attente)

On suppose être à l'intérieur d'une répétition approchée dont le taux de ressemblance entre copies $p$ est connu. On considère alors les distances entre les répétitions exactes de taille $\geq k$. On recherche une majoration de ces distances, ou plus exactement une majoration statistique (une vrai majoration serait théoriquement infinie).

En utilisant le modèle d'alignement binaire précédent, cela revient à étudier la distance entre les trains de $k$ piles. En supposant que l'on vienne de générer un train de $k$ piles, cette distance sera égale au nombre de lancers écoulés avant de pouvoir à nouveau générer un train de $k$ piles.

La variable aléatoire $G_{k,p}$ comptant ce nombre de lancers écoulés s'appelle le « temps d'attente ». La probabilité que ce nombre de lancers soit égal à $x$ est donnée par la formule récursive suivante.

$$\mathcal{P}\{G_{k,p} = x\} = \begin{cases} 0 & \text{pour } 0 \leq x < k \\ p^k & \text{pour } x = k \\ (1-p)p^k[1 - \sum_{i=0}^{x-k-1} \mathcal{P}\{G_{k,p} = i\}] & \text{pour } x > k \end{cases} \tag{7}$$

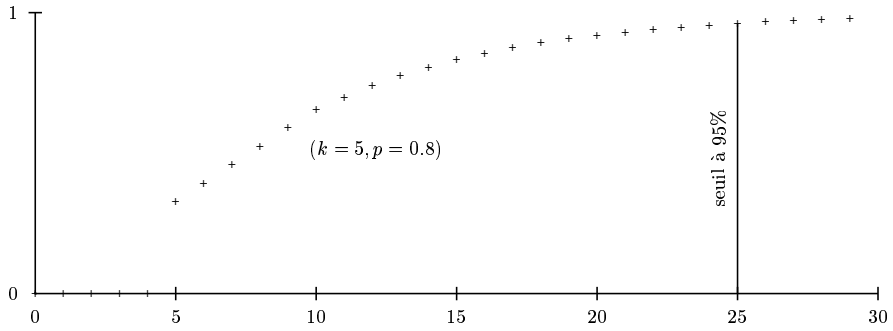



FIG. 5 – *Waiting Time distribution*

**Utilisation :** cette majoration statistique de la distance entre répétitions exactes de taille $\geq k$ dans une répétition approchée nous servira justement de critère de regroupement des répétitions exactes.

Par exemple, en prenant $\alpha = 5\%$, $p = 80\%$ et $k = 5$, le seuil calculé est alors de 25. En ayant deux répétitions exactes $A$ et $B$ dont les occurrences sont respectivement $A_1, A_2$ et $B_1, B_2$, on pourra alors décider si ces répétitions peuvent être

regroupées en vérifiant que les distances entre occurrences $d(A_1, B_1)$ et $d(A_2, B_2)$ (ou $d(A_1, B_2)$ et $d(A_2, B_1)$) soient inférieures à 25.

## 3.2 *Random Walk* (marche aléatoire)

Lorsqu'une insertion/suppression d'une lettre touche une zone de la séquence, elle a pour conséquence de faire varier les distances entre les lettres qui précédaient la zone modifiée et les lettres qui la succédaient.

Nous étudirons dans cette partie la distribution décrivant la variation de distance provoquée par les *indels*.

Cette variation est à la fois dépendante de la fréquence $p$ par lettre des *indels*, mais aussi de la taille $n$ du segment d'ADN sur lequel ces *indels* sont influents. On utilisera, pour modéliser cette variation de distance, une marche aléatoire monodimensionelle. Une marche aléatoire est un déplacement probabiliste dans un espace effectué à intervalles discrets. On définit à chaque étape trois comportements ayant une probabilité d'arriver :

- «*aller un pas vers la gauche*» avec probabilité $p$
- «*aller un pas vers la droite*» avec probabilité $p$
- «*rester sur place*» avec probabilité $1 - 2p$

Ces trois comportements sont respectivements similaires à une suppression unitaire d'une lettre, une insertion unitaire d'une lettre, ou un autre évenement (substitution ou conservation de lettres).

Au départ, le mobile se trouve à la position $0$. On s'intéresse aux probabilités concernant sa position $S_n$ au bout de $n$ étapes. En particulier, on aimerait déterminer dans quel intervalle minimum $[-L; L]$ (bornes statistiques) le mobile se trouve avec une probabilité fixé au bout de $n$ étapes.

### 3.2.1 Formule directe

Soit $k$ l'indice de la position finale du mobile au bout de $n$ étapes ($-n \leq k \leq n$), et soit $j$ le nombre de déplacements ayant été réalisés à gauche ($0 \leq j \leq (n-k)/2$), on déduit alors que le nombre de déplacements ayant été réalisés à droite est de $j + k$.

On déduit également que le nombre écoulé d'étapes sans se déplacer est de $n - (2j + k)$

La probabilité d'une telle combinaison de déplacements ($j$ à gauche $k + j$ à droite, $n - (2j + k)$ sur place) est donnée par une loi multinomiale :

$$\frac{n!}{j!(j+k)!(n-(2j+k))!} p^{2j+k}(1-2p)^{n-(2j+k)} \tag{8}$$

Pour déterminer la probabilité que le mobile soit à la position $k$ au bout de $n$ itérations, il faut sommer l'équation (8) sur les valeurs possibles de $j$ :

$$\sum_{j=0}^{(n-k)/2} \frac{n!}{j!(j+k)!(n-(2j+k))!} p^{2j+k}(1-2p)^{n-(2j+k)} \tag{9}$$

Enfin pour déterminer la probabilité que le mobile termine sa course dans l'intervalle $[-L, L]$, on effectue une sommation de l'équation (9) avec $k$ parcourant l'intervalle précédent :

$$\sum_{k=-L}^{+L} \sum_{j=0}^{(n-k)/2} \frac{n!}{j!(j+k)!(n-(2j+k))!} p^{2j+k}(1-2p)^{n-(2j+k)} \tag{10}$$

### 3.2.2 Formule génératrice

Le calcul de la formule (10) pose des problèmes d'*overflow* provoqués par le coefficient multinomial. Il existe cependant une astuce de calcul : elle consiste à utiliser un polynôme générateur afin de calculer cette même quantité.

Si l'on considère le polynôme $P(X) = pX + (1-2p) + p\frac{1}{X}$, on remarque que

$$P^n(X) = a_{-n}X^{-n} + a_{-n+1}X^{-n+1} + \cdots + a_{n-1}X^{n-1} + a_nX^n \tag{11}$$

a pour coefficients $a_i$ la probabilité que le mobile soit positionné en $i$ au bout de $n$ itérations.

En effet, le coefficient du monôme à la puissance $i$ de $P^n(X)$ évalue la somme des probabilités de tous les chemins terminant à la position $i$ au bout de $n$ itérations. Cela permet, en calculant les coefficients du polynôme $P^n(X)$, d'avoir une formule directe.

Lorsque $n$ est grand, une loi normale donne une bonne approximation de la variable aléatoire $S_n$. Le polynôme générateur permet alors de déterminer l'espérance et la variance de la loi.

$$\mu(S_n) = \sum_x \mathcal{P}\{S_n = x\} \cdot x = \frac{\partial P^n(X)}{\partial X}\Big|_{X=1} = 0 \tag{12}$$

$$\sigma^2(S_n) = \sum_x \mathcal{P}\{S_n = x\} \cdot x^2 = \frac{\partial}{\partial X}\Big(X \cdot \frac{\partial P^n(X)}{\partial X}\Big)\Big|_{X=1} = 2np \tag{13}$$

### 3.2.3 Bornes statistiques d'une marche aléatoire

On s'intéresse au *bornes statistiques* de la marche aléatoire ainsi générée : on veut pouvoir encadrer dans un intervalle [-L..L] la position de fin de course du mobile au bout de $n$ itérations avec une certitude donnée (en général de 95 %).

Lorsque la modélisation utilisée ne peut pas être approchée par une loi normale ($n$ trop petit), on doit rester en régime fini. On effectue alors une sommation des

probabilités calculés par génération de coefficients. Cette sommation s'effectue sur des intervalles [-L..L] croissants, jusqu'à atteindre le seuil des 95% .

On détermine ainsi, à partir des bornes statistiques de la marche aléatoire, la variation statistique maximale de distance due aux *indels* entre deux occurrences d'une répétition.

# 4 Liste chaînée des $k$-mots

Une première méthode à la base de nombreux algorithmes lorsqu'il s'agit de rechercher des graines consiste à créer une liste chaînée des $k$-mots (sous-mots de taille $k$) de la séquence $T$ à traiter. Dans une séquence $T$ de $n$ lettres, il y a donc $n-k+1$ $k$-mots commençant aux positions $1, 2, \cdots, n-k+1$.

L'algorithme présenté ici permet de construire une liste chaînée des $k$-mots. Il est divisé en deux étapes :

## 4.1 Codage des $k$-mots

La première étape consiste à coder ces $k$-mots dans un tableau d'entiers $K$ de taille $n-k+1$. On utilisera le codage naturel pour coder un $k$-mot $w$ par un entier $C(w) \in [0 : \mathcal{A}^k - 1]$ :

$$C(w) = \sum_{i=1}^{k} \mathcal{A}^{i-1} Val(w[i])$$

Ici,

- $\mathcal{A}$ est la taille de l'alphabet (quatre dans le cas de l'ADN)
- $w[i]$ désigne la $i^{me}$ lettre du $k$-mot à coder
- $Val()$ donne la valeur entière associée à une lettre ; dans le cas de l'ADN, on prendra arbitrairement la convention suivante :

$$\{\texttt{A}, \texttt{T}, \texttt{G}, \texttt{C}\} \xrightarrow{Val} \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}\}$$

## 4.2 Chaînage des $k$-mots

La deuxième étape génère une liste chaînée des positions des occurrences de chaque $k$-mot.

Pour cela, on crée un tableau $L$ de taille $n-k+1$ qui servira de liste chaînée : la valeur $L(i)$ du $i^{me}$ élément de $L$ (si elle est non nulle) désignera la position de l'occurrence suivante du $k$-mot situé en $i$. Cela vérifie, en terme de codage, $K(L(i)) = K(i)$.

Si l'on connaît la première position $i_0$ d'un $k$-mot donné, on déduit alors toutes les positions suivantes $i_j = L^j(i_0) = \underbrace{L(L \cdots L(i_0) \cdots)}_{j}$.

Un tableau $F$ de taille $\mathcal{A}^k$ sert à mémoriser, pour chaque $k$-mot $w$, la position $i_0$ de sa première occurrence à l'indice $C(w)$.

Enfin un tableau annexe $B$ également de taille $\mathcal{A}^k$ est nécessaire afin de mémoriser la position de la dernière occurrence trouvée du $k$-mot $w$.

Les trois tableaux précédents sont initialisés à 0, et les opérations données à la Table 2 sont réalisées.

TAB. 2 – Algorithme de chaînage des $k$-mots

```
1   pour j de 1 à n − k + 1 faire
2       c ← K(j) ; i ← B(c)
3       si (i = 0) alors
4           F(c) ← j
5       sinon
6           L(i) ← j
7       fsi
8       B(c) ← j
9   fpour
```

Remarquons que les deux tableaux $B$ et $F$ sont de taille exponentielle en fonction de $k$.

# 5 Algorithme

## 5.1 Description

L'algorithme proposé ici a pour but de trouver des répétitions approchées, ou plus exactement des zones pouvant contenir des répétitions approchées (zones candidates).

Le but de l'algorithme est dans un premier temps de retrouver des sous fragments répétés de manière exacte. Dans un deuxième temps, il cherchera à regrouper ces sous fragments afin de déterminer des zones candidates pouvant contenir une répétition approchée.

## 5.2 Principes

On utilisera par la suite les notations suivantes :

- un $k$-mot $w$ débutant à la position $i$ sur le texte sera noté $w_i$.
- un couple de $k$-mots identiques $(w_i, w_j)$ sera noté $c(i, j)$ (avec $j < i$ par convention).

L'algorithme prend comme entrée la liste chaînée des $k$-mots extraite du texte $T$ à traiter. (voir chapitre 4) Cette liste permet de déterminer, pour n'importe quel $k$-mot $w$, toutes les positions des occurrences de $w$ dans $T$.

On considérera alors les couples formés par 2 positions $i$ et $j$ sur le texte contenant le même $k$-mot ($j < i$).

Intuitivement, lorsqu'une répétition approchée existe, il serait intéressant de pouvoir trouver des ensembles de couples de $k$-mots

$$c(i_1, j_1), c(i_2, j_2), \ldots, c(i_x, j_x)$$

tels que les suites $i_1, i_2, \ldots, i_x$ et $j_1, j_2, \ldots, j_x$ soient strictement croissantes, relativement « condensées » (peu d'espace entre $i_y$ et $i_{y+1}$ ) et dont l'écart $i_y - j_y$ reste approximativement le même lorsque $y$ croît. On en déduirait alors que la suite constituée des indices $j$ se situerait à l'intérieur de la première occurrence de la répétition approchée tandis que la deuxième occurrence contiendrait la suite $i$.

## 5.3 Utilisation des critères statistiques

Nous considérerons d'abord que nous possédons à priori des connaissances statistiques sur les répétitions approchées recherchées. Nous connaissons en particulier le taux d'erreur maximal $p_M$ des copies de ces répétitions.

Il nous est alors possible de déterminer la distance statistique maximale entre 2 $k$-mots successifs, à l'aide de la distribution du *Waiting Time*(voir chapitre 3.1.2).

Nous connaissons par ailleurs la probabilité $p_I$ d'insertions/suppressions de lettres individuelles.

Les insertions/suppressions d'éléments modifient l'écart $i_y - j_y$ entre les deux suites. A l'aide de bornes statistiques sur une *Marche Aléatoire*, on peut évaluer également la variation tolérée de $i_y - j_y$ entre deux indices successifs. (voir chapitre 3.2)

### 5.3.1 Distances considérées

Pour évaluer les deux critères précédemment énoncés, deux distances doivent être considérées ; prenons deux couples $c(i', j')$ et $c(i, j)$.

- La distance inter-couples sera donnée par $i - i'$ plutôt que par $j - j'$ (c'est un choix à priori arbitraire mais qui nous garantira sa positivité plus tard lors de l'algorithme).
- La distance intra-couple sera la distance entre les deux $k$-mots de chaque couple ($i - j$ et $i' - j'$ respectivement)

Les deux premiers critères appliqués à ces deux distances permettent d'établir les deux principes de l'algorithme de chaînage des couples :

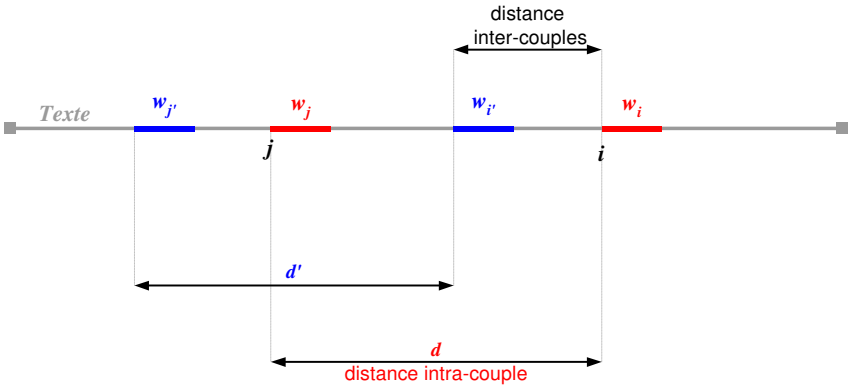



FIG. 6 – Distance inter-couples et intra-couple

Chaque couple $c(i,j)$ (couple de positions d'occurrences d'un même *k*-mot) est chaîné avec un couple $c(i',j')$ précédemment trouvé ($i' < i$) s'il satisfait les deux critères suivants :

1. la distance inter-couples calculée est inférieure à un seuil $\rho$ déterminé à partir de critères statistiques.
2. la variation de distance intra-couple entre deux couples successifs est, elle aussi, inférieure (en valeur absolue) à un seuil $\delta$ déterminé à partir de critères statistiques.

### 5.3.2 Première version

Voici une première approche de l'algorithme sur un texte $T$ de taille $n$.

TAB. 3 – Algorithme de chaînage des couples de $k$-mots (1)

```
1   pour chaque k-mot w_i de T (1 ≤ i ≤ n − k + 1) faire
2       pour chaque occurrence w_j de w_i (i < j) faire
3           si il existe un couple c(i', j') satisfaisant
4                   les deux critères
5           alors
6                   chaîner c(i', j') vers c(i, j)
7           fsi
8       fpour
9   fpour
```

L'algorithme parcourt pour $i \in [1..n-k+1]$ chaque *k*-mot $w_i$ du texte $T$ . Il

considère alors (à l'aide de la liste chaînée des *k*-mots $L$ ) chacune des occurrences précédentes $w_j$ du *k*-mot $w_i$ (on a donc $w_j = w_i$ et $j < i$).

On forme ainsi un couple $c(i, j)$ ayant une distance intra couple $d = i - j$. Ce couple ne peut être chaîné à un autre couple que s'il satisfait les deux conditions énoncées auparavant.

Le problème principal pour évaluer ces critères est d'avoir pu mémoriser suffisamment d'information sur les couples précédemment trouvés.

## 5.4 Respect des critères

### 5.4.1 Méthode adoptée

Pour pouvoir vérifier ces deux critères, on utilise un tableau des distances. Le rôle de ce tableau est de **conserver, à l'indice $d$, la position $i$ du dernier couple dont la distance intra-couple était $d$.**

Ce tableau nommé $CD$ de taille $n - k + 1$ sera remis à jour après chaque découverte d'un nouveau couple.

Supposons que l'on vienne de découvrir un nouveau couple $c(i, j)$, sa distance intra-couple est donc $d = i - j$. Avant de mettre à jour le tableau des distances, on va considérer les couples précédemment trouvés qui ont une distance intra couple $d_{obs}$, soit égale à $d$, soit proche de $d$ (la variable $d_{obs}$ va en fait parcourir l' intervalle $[d - \delta..d + \delta]$)

- Si parmi ces couples, l'un d'entre eux satisfait le critère de distance inter couples, il sert alors de support de chaînage pour $c(i, j)$.
- Dans le cas contraire, le couple n'est pas chaîné mais on met tout de même à jour le tableau des distances.

Afin de trouver un couple précédent satisfaisant le critère de distance inter couples tout en respectant le mieux celui de distance intra couple, on considère les indices $d_{obs}$ dans la fenêtre $[d - \delta, d + \delta]$ en les parcourant en « zigzag » ($d, d + 1, d - 1, d + 2, \ldots, d + \delta, d - \delta$).

- Si l'on trouve une valeur $d_{obs}$ telle que $i - CD[d_{obs}] < \rho$, le couple $c(i, j)$ peut alors être chaîné au couple $c(i', j')$ ( avec $i' = CD[d_{obs}]$ et $j' = i' - d_{obs}$ ) et l'on arrête le parcours $d_{obs}$.
- Le cas échéant, on n'oubliera pas de remettre à jour $CD[d]$ avec la valeur $i$.

### 5.4.2 Deuxième version

Voici une version complétée de l'algorithme.

## 5.5 Gestion des couples

Un problème non encore abordé ici concerne la gestion des couples. Les couples, lorsqu'ils sont instanciés, sont conservés dans des chaînages eux même conservés dans une liste chaînée.

TAB. 4 – Algorithme de chaînage des couples de $k$-mots (2)

```
01    pour chaque k-mot w_i de T (1 ≤ i ≤ n − k + 1) faire
02        pour chaque occurrence w_j de w_i (i < j) faire
03            d ← i − j
04            pour d_obs ∈ {d, d + 1, d − 1, ..., d + δ, d − δ} faire
05                i' ← CD[d_obs]
06                si i − i' < ρ alors
07                    j' ← i' − d_obs
08                    chaîner c(i', j') vers c(i, j)
09                    break //sortir de la boucle d_obs
10                fsi
11            fpour
13            CD[d] ← i
13        fpour
14    fpour
```

**Création d'instance de couples**

Un couple $c(i, j)$ qui ne peut pas être chaîné à des couples précédemment trouvés n'est pas instancié. On le conserve cependant dans le tableau des distances ( instruction $CD[d] \leftarrow i$ ).

La création d'instances n'est possible que si au moins deux couples respectent les deux critères. Dans ce cas, si l'ancien couple était déja instancié, le nouveau sera logiquement instancié et chaîné à sa suite. Si l'ancien couple n'avait pas été instancié (car il ne pouvait pas être chaîné à des couples précédents), on va alors créer un nouveau chaînage contenant une instance de l'ancien couple suivie d'une instance du nouveau couple.

**Accès aux instances de couples**

Afin d'avoir accès à l'instance du dernier couple dont la distance est $d_{obs}$, un tableau de pointeurs $CDC$ (au fonctionnement similaire à $CD$) sera utilisé. Il indiquera pour chaque distance intra couple $d$, l'instance du dernier couple dont la position est donné par $CD(d)$, si ce dernier est effectivement instanciée.

**Libération d'instances inutilisés**

On remarque qu'une telle liste peut prendre en mémoire une ampleur très importante. Cela est d'autant plus regrettable que des chaînages anciens (tous les couples de ces derniers ont une valeur $i' < i_{courant} - \rho$), n'ont aucune chance d'être prolongés par les couples découverts.

Il s'avère alors intéressant de libérer « périodiquement » les chaînages inutilisés par l'algorithme, après les avoir traités avec un algorithme d'alignement de

séquences par exemple.

**Optimisations**

Ces trois optimisations ont un intérêt pratique et peuvent donc être laissées lors d'une première lecture.

1. Une répétition exacte de taille $r > k$ va donner lieu à une succession de couples $c(i_1, j_1), c(i_1 + 1, j_1 + 1), \ldots, c(i_1 + r - k, j_1 + r - k)$.

   On peut éviter cette multiplication inutile de couples en conservant une information supplémentaire $size$ donnant la taille de la répétition avant les indices $i_1, j_1$ : elle signifira qu'il existe une répétition entre le mot $[i_1 - size, i_1 + k - 1]$ et le mot $[j_1 - size, j_1 + k - 1]$.

   On vérifiera lors de l'ajout à un chaînage si le couple précédent est $c(i - 1, j - 1, size)$, auquel cas ce dernier deviendra à $c(i, j, size + 1)$

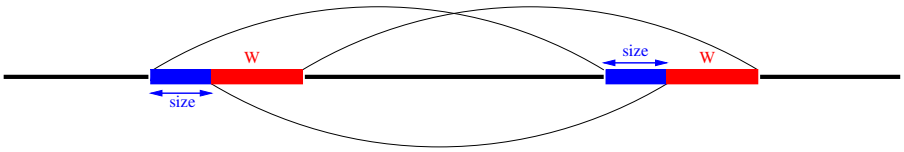



FIG. 7 – Extension des $k$-mots lors de répétitions exactes

2. Certaines répétitions en tandem provoquent des profusions de couples. Le problème provient surtout de l'exposant de ces répétitions qui, lorsqu'il est fort (et que la période est faible), génère de nombreux couples.

   Si la période de la répétition est de taille 1 ($ie$ la même lettre se répète sur une séquence de taille $> k$), on peut éviter ce désagrément en considérant par couple une valeur de recouvrement à droite ($recouvrement\ gauche$) et à gauche ($recouvrement\ droit$).

   Ces valeurs signifient que, pour le couple de $k$-mots $c(i, j)$, les $k$-mots à gauche $w_j, w_{j-1}, w_{j-recouvrement\ gauche}$ et les $k$-mots à droite $w_i, w_{i-1}, w_{i-recouvrement\ droit}$ sont les mêmes.

   Cette optimisation ne joue alors plus aucun rôle si la répétition a une période $\geq 2$ et le problème est alors ouvert.

3. Lorsque l'on trouve un couple $c(i', j')$ satisfaisant les critères, ce dernier a quelquefois déjà un ou plusieurs successeurs, il est alors intéressant de tester les deux optimisations à ce couple et le cas échéant à ces successeurs avant de l'ajouter en fin de chaînage.

# 6 Comparaison avec l'algorithme de BENSON

Bien que notre algorithme soit destiné à la recherche de répétitions distantes, certaines caractéristiques de l'algorithme de BENSON (dedié à la recherche de répétitions en tandem) nous ont inspirés. Nous indiquons ici les quelques points où les algorithmes concordent, et les différences principales qui les distinguent.

- les notions de $k$-mot, de couple de $k$-mots restent toujours présentes. La création de la liste chaînée des $k$-mots reste la même (chapitre 4).
- certains des critères utilisés par Benson ont été repris, comme la marche aléatoire pour simuler les *indels*[5] (chapitre 3.2) bien que la méthode de calcul ait été réalisé de manière exacte à l'aide d'une fonction génératrice. D'autres critéres ont été ajoutés comme la distance statistique entre $k$-mots (chapitre 3.1.2), la taille espérée maximale d'un $k$-mot (chapitre 3.1.1).

Les critères ajoutés nous ont permis de faire des choix différents en ce qui concerne les structures de données.

- un tableau des distances a été choisi plutôt qu'une liste des distances (voir chapitre 5.4.1).
- Benson utilisait une liste chaînée pour stoquer des couples de $k$-mots dont il aurait éventuellement besoin pour valider une répétition en tandem. Les chaînages de couples générés par notre approche sont déjà considérés comme des zones pouvant contenir des répétitions approchées. Ils sont construits de manière continue en utilisant un seuil sur la distance statistique entre $k$-mots (voir chapitre 5.5).
- La recherche de répétitions ne s'effectue alors pas sur la liste complète mais sur ces chaînages durant le déroulement de l'algorithme pour valider ou réfuter l'hypothèse d'une répétition approchée.

# 7 Réalisation

Le programme de chaînage des $k$-mots a été réalisé en C ANSI et testé sur un Pentium II 350 disposant de 64 Mo de Ram.

Il est constitué de 8 fichiers contenant le code source propre au programme et de 7 fichiers d'entête répartis comme suit :

- `assemble.h` et `assemble.c` implémentent l'algorithme de chaînage des couples de $k$-mots. Cet algorithme est réalisé en deux exemplaires permettant de rechercher des ressemblances sur une seule et respectivement deux séquences d'ADN.
- `align.h` et `align.c` permettent d´évaluer un alignement sur les chaînages des couples de $k$-mots terminés.
- `prdyn.h` et `prdyn.c` contiennent des algorithmes de programmation dynamique destinés à l'alignement de sous-fragments de texte.

[5] insertions et suppressions d'éléments

- `tuple.h` et `tuple.c` servent à la gestion des couples de $k$-mots et des chaînages de couples.
- `proba.h` et `proba.c` implémentent toutes les méthodes relatives aux probabilités, permettant par exemple de calculer le *Waiting Time*, les bornes statistiques d'une marche aléatoire, etc.
- `kword.h` et `kword.c` sont utilisés pour la construction de la liste des $k$-mots, et plus généralement tout ce qui se rapporte à la gestion des données.
- `util.h` et `util.c` proposent un certain nombre de macros et d'opérateurs génériques.
- `main.c` contient l'interface du programme.

L'ensemble totalise environ 2300 lignes de code.

Les positions des occurrences répétées et la qualité de l'alignement obtenu sont données sur la sortie standard. Il est par ailleurs possible de visualiser les alignements obtenus sous la forme suivante :

```
ttcttgtctt-tcatgtacct-ctttcagatacc--actgagtaatatgacttta-aaagctct
......d.s.i..sd......i.ss.d....s.sii...ss...s.s..d....si...ssd..
ttcttg-catatcc-gtacctaccgt-agattcaatactccgtagtttg-ctttcgaaata-ct
```

Les opérations réalisées sur deux fragment d'ADN à comparer sont alors indiquées par des lettres dans la ligne centrale («`d`» pour un suppression d'une nucléotide, «`i`» pour une insertion d'une nucléotide, et «`s`» pour une substitution de nucléotides).

# 8 Expérimentation

Afin d'évaluer les performances du programme en terme de vitesse et d'efficacité à retrouver des répétitions approchées, deux chromosomes de levure (il s'agit du chromosome `V` et du chromosome `IX` de *S.cerevisiae*[6]) ont été comparés. Ils totalisent respectivement 576869 et 439885 nucléotides.

Des résultats d'`ASSIRC` sur ces deux chromosomes étaient disponibles dans l'article « *A strategy for finding regions of similarity in complete genome sequences* » [VBA+98].

Par ailleurs, le logiciel `BLASTN` a aussi été utilisé pour réaliser une étude comparative.

## 8.1 Performances

Le tableau ci-dessous donne quelques performances du programme selon la taille des $k$-mots.

[6] disponibles à l'URL *http ://genome-www4.stanford.edu/cgi-bin/SGD/seqTools*

Le seuil $\rho$ sur la distance inter-couples et le seuil $\delta$ sur la variation de distance intra-couple calculés et utilisés par l'algorithme sont donnés (probabilité de mutation de 30%, probabilité d'*indels* de 20%, taux d'erreur $\alpha$ de 5%). Le nombre de chaînages candidats à un alignement, le nombre de répétitions trouvées, ainsi que le temps de calcul y sont indiqués.

| $k$ | $\rho$ | $\delta$ | nb.chaînages | output | time |
|---|---|---|---|---|---|
| 8 | 152 | 11 | 33619 | 67 | 1 min |
| 7 | 102 | 9 | 63734 | 60 | 2 min |
| 6 | 82 | 7 | 457216 | 66 | 8 min |
| 5 | 58 | 6 | 2733769 | 65 | 35 min |
| 4 | 40 | 5 | 15873607 | 69 | 161 min |

TAB. 5 – Performances du programme selon la taille des $k$-mots

La taille de la sortie varie à cause d'un phénomène de fragmentation des chaînages que nous expliquons plus loin.

Un indice intéressant concerne le nombre de secondes de calcul partagées entre l'algorithme de chaînage (noté par la suite $T_C$) et celui d'alignement des chaînages ($T_A$).

| | $p_M = 10\%$ | | | | $p_M = 33\%$ | | |
|---|---|---|---|---|---|---|---|
| $k$ | $T$ | $T_C$ | $T_A$ | $k$ | $T$ | $T_C$ | $T_A$ |
| 9 | 4 | 3 | 1 | 9 | 213 | 6 | 207 |
| 8 | 13 | 10 | 3 | 8 | 531 | 20 | 511 |
| 7 | 47 | 36 | 11 | 7 | 612 | 56 | 555 |
| 6 | 171 | 126 | 45 | 6 | 1374 | 185 | 1189 |
| 5 | 655 | 442 | 213 | 5 | 2727 | 921 | 1806 |

TAB. 6 – Temps de calcul partagé entre le chaînage et l'alignement

On remarque que, lorsque le paramètre $p_M$ augmente, le temps de calcul consommé par l'algorithme d'alignement croît de manière beaucoup plus importante que celui consommé par l'algorithme de chaînage.

En effet, le temps de chaînage double approximativement entre les deux tableaux, quel que soit la taille des $k$-mots, alors que taux de multiplication du temps de calcul consommé par l'algorithme d'alignement semble suivre, une courbe croissante de paramètre $k$.

Il serait donc intéressant, pour gagner en efficacité, de pouvoir améliorer d'abord l'algorithme d'alignement.

## 8.2 Comparaison avec ASSIRC

Les séquences ayant plus de 75% de ressemblance sont toutes trouvées par l'algorithme en moins d'une minute avec des paramètres standards ($k = 8$, $20\%$ d'*indels*, $25\%$ de mutations). Elles ont un taux de ressemblance sensiblement plus élevé que celui donné par ASSIRC et sont donc mieux localisées.

En fixant des paramètres statistiques plus élevés et en utilisant des $k$-mots plus petits, ($k = 5$, taux de mutation de 30%, et taux d'*indels* de 30%), seules deux séquences n'ont pu être retrouvées après 15 minutes de calcul (notons que ASSIRC a passé environ 17 heures pour trouver de tels résultats).

L'une d'entre elles (possédant une graine maximale de taille 7 seulement) est qualifiée de 62.9 % de ressemblance, mais après alignement, elle ne totalise pourtant que 58% de ressemblance entre copies sur nos séquences.

En contrepartie, de nombreuses autres répétitions de meilleure qualité ont pu être trouvées qui ne figuraient pas dans le tableau récapitulatif d'ASSIRC

## 8.3 Comparaison avec BLASTN

BLAST est un logiciel d'alignement de séquences décliné sous de nombreuses versions. Le logiciel BLASTN (N comme *nucléotide*) a été installé et utilisé pour le test comparatif : il trouve, à temps de calcul équivalent, plus de répétitions mais beaucoup ont un taux de qualité relativement faible.

Il possède cependant l'inconvénient de rater des répétitions dont les graines maximales sont de taille trop petite, ce qui devient fréquent lorsque, par exemple, le taux de ressemblance demandé est faible ($< 70\%$).

Certaines répétitions que nous déterminons avec des taux de similarité de plus de 70% (Tableau) ne sont alors pas trouvées par BLASTN.

| séquence 1 | séquence 2 | err. | gr. max | gr. min | nb de gr. |
|---|---|---|---|---|---|
| 270787-271078 | 257894-258185 | 73 | 13 | 10 | 24 |
| 273624-273841 | 259542-259759 | 61 | 13 | 10 | 15 |

TAB. 7 – Répétitions caractérisées contenant de petites graines

Pourtant BLASTN nous signale qu'il arrive à déterminer des répétitions ayant de plus faibles taux de ressemblance.

## 8.4 Fragmentation des répétitions

Ce problème particulier de l'algorithme est qu'il lui arrive quelquefois de « fragmenter » les répétitions, c'est à dire de donner deux répétitions de taille inférieure lorsque d'autres algorithmes ne trouvaient en fait qu'une seule répétition.

Cela est dû à la variable $\rho$ sur la distance entre $k$-mots : lorsque cette distance est supérieure au seuil calculé (c'est à dire dans 5% des cas), les $k$-mots ne sont pas chaînés et les répétitions apparaissent fragmentées.

Il est possible d'éviter cela en diminuant le taux d'erreur $alpha$. Cette solution a alors tendance à augmenter le seuil et donc à ralentir l'algorithme.

Il serait envisageable, plutôt que de considérer des distances plus importantes, de traiter le problème à la sortie, en essayant de rassembler à posteriori des répétitions qui auraient été « coupées » par inadvertance.

## 8.5 Extensions envisagées

### 8.5.1 Sous $k$-mots

Le coût de l'alignement est le principal facteur de ralentissement de l'algorithme lorsque ce dernier est lancé avec des paramètres relativement fins.

Une idée qui permettrait de filtrer les chaînages créés avant d'utiliser les méthodes d'alignement classiques serait de réitérer en quelque sorte l'algorithme sur les zones candidates avec une taille de $k$-mot $k_2$ plus petite.

En effet, le coût en temps de calcul d'un alignement entre deux $k$-mots dépend uniquement de la distance $d$ entre ces $k$-mots : ce coût est de $d^2$.

En choisissant de réitérer, sur l'intervalle entre ces $k$-mots, un algorithme de chaînage de couples de $k_2$-mots ($k_2 < k$), la complexité par rapport au problème précédent est alors divisée par $\mathcal{A}^{k_2}$ où $\mathcal{A}$ désigne la taille de l'alphabet (4). On passe alors à un algorithme d'une complexité $d^2/\mathcal{A}^{k_2}$.

Des critères statistiques sur le nombre de $k_2$-mots permettraient alors d'éliminer les chaînages dont on est quasiment sûr qu'ils ne satisferont pas les critères de ressemblance. Ces mêmes $k_2$-mots permettraient également d'estimer globalement le taux de ressemblance, et de servir de points d'ancrage à des algorithmes d'alignement, diminuant ainsi considérablement la complexité de l'alignement final effectué pour évaluer le taux de ressemblance entre copies.

### 8.5.2 Chaînes Inversées et Complémentaires

Il arrive que des répétitions apparaissent dans le texte, mais dans des sens opposés. Il serait alors intéressant de pouvoir traiter non seulement ce cas, mais aussi le moins évident et pourtant tout aussi intéressant problème des bases complémentaires.

En effet, pour une chaîne d'ADN donnée, on ne traite en fait qu'un seul brin, par exemple `AGTGCATG...`. Le brin complémentaire, qui ne nous est pas fourni, mais qui est parfaitement connu car formé par une substitution des paires de bases `A` $\leftrightarrow$ `T` et `G` $\leftrightarrow$ `C`,(`TCACGTAC...`) n'est alors pas traité, ce qui peut être un handicap s'il y a eu des échanges d'ADN entre les brins.

# 9 Conclusion

Nous avons présenté une nouvelle méthode de recherche de répétitions approchées basée, d'une part sur des propriétés statistiques des séquences, et d'autre part

sur un algorithme approprié de chaînage des couples de $k$-mots.

L'approche proposée utilise un modèle probabiliste afin de déterminer des critères permettant de rassembler les graines, plutôt que de suivre le concept couramment utilisé par d'autres algorithmes qui consiste simplement à étendre les occurrences de répétitions exactes.

Les tests qui ont été réalisés montrent que l'idée sous jacente exploitée (ie, le regroupement de graines pour la recherche de répétitions approchées) permet d'aboutir à une solution satisfaisante du point de vue du temps de calcul mais aussi de la qualité du résultat final.

Nous avons enfin proposé quelques améliorations sur le plan théorique et pratique qui sont susceptibles de rendre l'algorithme plus efficace qu'il ne l'est actuellement.

# Table des figures



# Liste des tableaux

## Références

[ACR99] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Oracle des facteurs, oracle des suffixes. Rapport i.g.m. 99-08, Université de Marne-la-Vallée, 1999. 6

[BBK91] B.Edwin Blaisdell, Chris Burge, and Samuel Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221 :1367–1378, 1991. 1, 7

[Ben98] Gary Benson. An algorithm for finding tandem repeats of unspecified pattern size. In *RECOMB 98*, pages 20–29, 1998. 9

[Ben99] Gary Benson. Tandem repeats finder : a program to analyse DNA sequences. *Nucleic Acids Research*, 27(2) :573–580, 1999. 9

[Bro99] T A Brown. *Genomes*. Bios Scientific Publishers, 1999.

[BS98] Gary Benson and Xiaoping Su. On the distribution of $k$-tuple matches for sequence homology : A constant time exact calculation of the variance. *Journal of Computational Biology*, 5(1) :87–100, 1998.

[CHL01] Maxime Crochemore, C. Hancart, and Thierry Lecroq. *Algorithmique du texte*. Vuilbert, 2001.

[DEKM98] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[GK00] Mathieu Giraud and Gregory Kucherov. Maximal repetitions and applications to DNA sequences. In *Journées Ouvertes : Biologie, Informatique et Mathématiques*, pages 165–172, Montpellier, May 3-5 2000. 8

[GSW86] Louis Gordon, Mark F. Schilling, and Michael S. Waterman. An extreme value theory for long head runs. *Probability Theory and Related Fields*, 72 :279–287, 1986. 10

[Jag01] Peter Jagers. Biological sequence analysis - conceptual, mathematical and statistical aspects, autumn 2001. 10

[KA90] Samuel Karlin and Stephen F. Altschul. Methods for assessing the statistical significance of molecular sequence feature by using general scoring schemes. In *Proc. Natl. Acad. Sci.*, volume 87, pages 2264–2268, 1990. 5

[KK99] Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 596–604, New-York (USA), 1999. IEEE Computer Society. 8

[KO87] Samuel Karlin and Friedemann Ost. Count of long aligned word matches among random letter sequences. In *Advances in Applied Probability*, volume 19, pages 293–351. 1987.

[KO88] Samuel Karlin and Friedemann Ost. Maximal length of common words among random letter sequences. In *Ann. Prob.*, volume 16, pages 535–563, USA, 1988.

[KOSS01] Stefan Kurtz, Enno Ohlebusch, Chris Schleiermacher, and Jens Stoye. Reputer : the manifold applications of repeat analysis. *Nucleic Acids Research*, 29(22) :4633–4642, 2001. 5

[KS99] Stefan Kurtz and Chris Schleiermacher. Reputer : fast computation of maximal repeats in complete genome. *Bioinformatics*, 15(5) :426–427, 1999. 5

[Mer99] Sabine Mercier. *Statistique des scores pour l'analyse et la comparaison de séquences biologiques*. PhD thesis, Université de Rouen, décembre 1999. 5

[MTV97] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. Serie of publications c, report c-1997-15, Department of computer sciences, 1997.

[Nic97] Pierre Nicodème. *Alignement avec des Familles de Séquences Protéiques*. PhD thesis, Université de Paris VII Jussieu, septembre 1997. 5

[PW95] Pavel Pevzner and Mickael Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, pages 135–154, 1995.

[Res] Wolfram Research. Mathworld . source web. `http ://mathworld.wolfram.com/Run.html`.

[VBA+98] Pierre Vincens, Laurent Buffat, Cécile André, Jean-Paul Chevrolat, Jean-François Boisvieux, and Serge Hazout. A strategy for finding regions of similarity in complete genome sequences. *Bioinformatics*, 14(8) :715–725, 1998. 8, 22

[VHS01] Natalia Volfovsky, Brian J Hass, and Steven L Salzberg. A clustering method for repeat analysis in DNA sequences. *Genome Biology*, 2(8), 2001.