N° d'ordre : 11/2008-M/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERHE SCIENTIFIQUE

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

Faculté d'Electronique et Informatique

MEMOIRE
Présenté pour l'obtention du diplôme de MAGISTER
EN : INFORMATIQUE

Spécialité : Intelligence Artificielle et Bases de Données Avancées

Par : AMER-EL-KHEDOUD Ismaïl

Thème

Nouveaux algorithmes pour le problème de la plus longue sous-séquence commune à deux séquences

Soutenu le 27/10/2008, devant le jury composé de :

Mme.AISSANI.Aïcha, Professeur	Président du Jury
Mme.BENSAOU.Nacéra, Maitre de conférences	Directeur de mémoire
M.BABA ALI.Riadh, Maitre de conférences	Examinateur
M.ABDEDDIM.Saïd, Maitre de conférences	Examinateur

Dédicaces

Je dédie cette thèse à :
Mes très chers parents.
Mes sœurs et mon frère.
Mes amis : Hamad, Amine, Naïma, Wassila, Idir, Samir, Nassim, Hocine, Fadhéla, Sabrina, Julie, Achour et à tous ceux qui m'aiment.
Aux amis des forums : PhilatelieDz, ALGERMARO, Doc'sCity, The others, Smile, Douar.

A.Ismaïl

Remerciements

Avant de présenter ce travail, je tiens à remercier Dieu tout puissant, de m'avoir donné le courage et la patience pour arriver à ce niveau d'études.

Mes remerciements s'adressent également, à :

- Mes professeurs qui ont suivi mes travaux : M.S.ABDEDDAIM et Mme.N.BENSAOU pour leurs aides, et précieux conseils.
- Membres du jury, d'avoir accepté de juger ce travail.

Je tiens aussi avec une extrême sincérité, à remercier ceux qui m'ont aidé, et soutenu a cours de mes études.

Sommaire

Abréviation

ABR	DESIGNATION
INDEL	Insertion-Deletion
PLCS	Longest Common Subsequence
LD	Distance de Livenstein

Liste des Tableaux

Liste des figures

Introduction générale

Introduction générale

L'un des domaines scientifiques qui actuellement se développe très rapidement est la biologie moléculaire. La bioinformatique est la science d'application des méthodes informatiques sur des problèmes de la biologie, en particulier celles liées aux données biologiques tels l'ADN, l'ARN et les protéines.

Sachant que ces données subissent durant leurs processus d'évolution des mutations, les biologistes sont alors poussés à chercher ce qui a changé dans ces données. Cette recherche consiste à comparer ces données, généralement sous formes de séquences.

Dans ce rapport nous nous intéressons à la comparaison de deux séquences d'ADN pour arriver à trouver la plus longue sous-séquence commune à ces deux séquences. Ce problème est communément appelé « the PLCS-problem» (the longest common subsequences problem).

Plusieurs algorithmes ont été proposés par Needleman et Wunsch(1970), Smith et Waterman(1981), Hunt et Szymanski(1977) et bien d'autres dont celui que nous exposerons dans ce travail, proposé par J.Y.Guo et F.K.Hwang dans [Guo05].

Afin de mener à terme ce travail, notre rapport est structuré comme suit :

Chapitre 1 : Nous commencerons notre exposé par une introduction au domaine de la bioinformatique et la définition des notions de base inhérent à la théorie des langages et la théorie des ensembles.

Chapitre 2 : Ce chapitre comporte l'ensemble des définitions et méthodes de calcul de la complexité spatiale et temporelle nécessaires à l'évaluation des performances d'un algorithme.

Chapitre 3 : Nous présenterons dans le troisième chapitre quelques algorithmes d'alignement de séquences, par l'approche de programmation dynamique et l'approche de programmation non-dynamique.

Chapitre 4 : Ce chapitre constitue la partie centrale de notre travail. Nous y exposons l'implémentation de l'algorithme du « primal-dual » par Pevzner et Waterman et celle améliorée de Hwang et Guo, ainsi que les conclusions que nous avons tirées de cette étude.

Chapitre 5 : Dans cette partie nous exposerons les deux applications que nous avons développé et qui sont l'implémentation de l'algorithme de Hwang et Guo et l'implémentation de Smith et Waterman, ainsi que les résultats obtenus suite à des tests sur des séquences.

Chapitre 6: Ce dernier chapitre propose les éventuelles extensions que nous pouvons apporter à l'algorithme de Hwang et Guo ainsi que les perspectives que peut représenter cette branche de recherche.

Une conclusion générale termine ce travail.

Chapitre I

Rappels et définitions

Chapitre I : Rappels et définitions.

Ce chapitre introduit les notions de bases de la bioinformatique, la théorie des langages et la théorie des ensembles. Nous commencerons par donner des définitions liées à la biologie tels l'ADN et les séquences nucléiques sur lesquelles nous appliquerons des algorithmes d'alignement. Nous enchaînerons ensuite avec quelques rappels sur la théorie des langages pour connaître les termes utilisés dans l'algorithmique du texte, et nous clôturerons par des rappels inhérents à la théorie des ensembles que nous utiliserons dans l'algorithme du Primal-Dual[Guo05]

I.1 Notions de bioinformatique :

Toutes les caractéristiques d'un organisme vivant sont déterminées par le type de protéine que fabrique cet être vivant. Pour vivre, une cellule doit donc pouvoir fabriquer les nombreuses protéines dont elle a besoin (protéines de structure, hormones, enzymes, etc.).
Cependant, pour fabriquer une protéine donnée, la cellule doit posséder deux choses :

• Des acides aminés (ce sont les pièces de construction).
• L'information, la « recette », permettant d'assembler les acides aminés dans le bon ordre, c'est-à-dire l'information lui indiquant quels acides aminés utiliser et l'ordre dans lequel ils doivent être liés les uns aux autres pour former la protéine.

On sait, depuis les années 40, que c'est l'ADN contenu dans le noyau qui indique à la cellule comment fabriquer toutes les protéines qu'elle doit synthétiser. Pour chacune des milliers de protéines fabriquées par une cellule, il y a, dans le noyau de la cellule, l'information, « la recette », permettant de fabriquer cette protéine. Le noyau des cellules humaines contiendrait ainsi, sous forme d'ADN, environ 30 à 50 000 recettes différentes de protéines. On appelle « gène » chacune de ces recettes.

I.1.1- Les nucléotides : [Win99]

Les nucléotides sont des éléments formés par l'association d'un « sucre », c'est-à-dire une molécule de glucide (le désoxyribose), d'un acide (l'acide phosphorique) et d'une « base azotée ». Il existe quatre bases différentes : adénine (A), thymine (T), cytosine (C) et guanine (G).

I.1.2 – La molécule d'ADN : [Win99]

L'ADN est une chaîne résultant de la polymérisation de nucléotides porteurs des quatre types de bases. C'est l'ordre d'enchaînement des bases qui détermine l'information génétique codée. Les polynucléotides sont des polymères de très grande taille.

I.1.3- La protéine : [09]

Une protéine est une macromolécule présente chez tous les êtres vivants. Les protéines sont fabriquées par nos cellules à partir de l' ADN et grâce au code génétique. De celles qui constituent nos cheveux (la kératine) à celles qui nous défendent contre les microbes (les anticorps), les protéines ont des fonctions très variées.

I.1.4- La molécule d'ARN : [Win99]

L'ARN sert d'intermédiaire dans la circulation de l'information génétique de l'ADN aux protéines. Bien que chaque gène d'une molécule d'ADN emmagasine les instructions codées pour la synthèse d'une protéine spécifique, il ne fabrique pas réellement la protéine. Le gène dirige plutôt la synthèse d'un type d'ARN appelé ARN messager (ARNm). La molécule d'ARNm interagit avec la machinerie de la synthèse protéique pour diriger la production d'un polypeptide.

I.1.5- Code génétique [Win99] :

Le code génétique décrit comment la séquence des bases est convertie en une séquence d'acides aminés au cours de la synthèse protéique. Ce code associe à chaque triplet de base le long d'un brin d'ADN un acide aminé (élément de base des protéines). La succession des bases d'une portion d'ADN détermine la succession des acides aminés composant la protéine qui correspond à cette portion d'ADN.

Les mots du langage génétique, appelés codons, ont tous la même longueur. Ce sont des triplets de nucléotides A, C, T (ou U) ou G. Mathématiquement, nous avons 43=64 combinaisons possibles pour ces quatre lettres en triplets. Parmi ces codes, ils existent trois qui désignent la fin de la traduction (codon stop). Aux 61 autres codons ne correspondent que 20 acides aminés. Le code génétique est donc dit dégénéré ou redondant car en moyenne, un acide aminé est codé par trois codons (appelés « codons synonymes »). Ainsi, en moyenne une mutation génétique sur trois affectant une séquence d'ADN codante n'entraîne aucune modification de la protéine traduite.

I.1.6- Circulation de l'information génétique : [Win99]

La relation entre l'ADN, l'ARN et les protéines est illustrée par la figure 1.1.

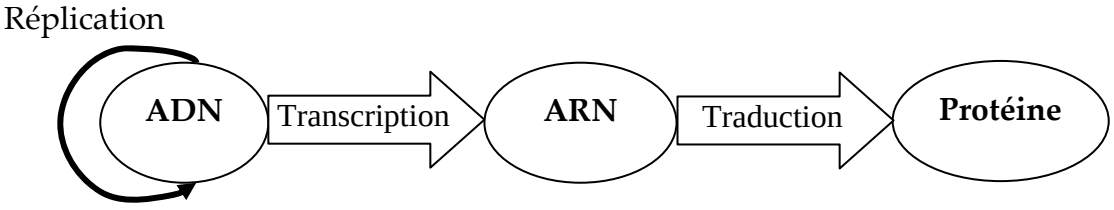


Figure 1.1 : Relation entre ADN, ARN et protéine.

Les flèches montrent le sens de circulation de l'information génétique.

Le processus qui duplique la molécule d'ADN, lorsque la cellule est divisée, est appelé «Réplication d'ADN ».

Le processus de création de la protéine peut être divisé en deux étapes :

Dans la première étape, l'information dans l'ADN est copiée dans le brin d'ARN avec la même séquence de nucléotides que dans le brin d'ADN. Cette étape est appelée « la transcription ».

La deuxième étape est appelée « traduction ». Un acide aminé dans la chaîne de la protéine est codé par trois nucléotides successifs dans la chaîne d'ARN, appelé « Codon ». Il existe 64 combinaisons de trois nucléotides mais il existe seulement 20 acides aminés. Donc plusieurs codons codent le même acide aminé, mais un codon ne code jamais plus d'un acide aminé.

I.1.7- L'évolution de l'ADN : [Win99]

L'ADN renferme les caractéristiques génétiques de l'espèce transmises de génération en génération grâce à la reproduction. On peut donc dire qu'il est porteur et garant de la stabilité des caractères au cours du temps. Mais il est également au cœur même du processus évolutif et des changements qui déterminent l'apparition de nouvelles espèces.

Mutations et réarrangements des chromosomes sont autant d'événements aléatoires qui modifient plus ou moins profondément les gènes. Ces changements sont ensuite soumis à la sélection naturelle pour être retenus ou rejetés par l'évolution. C'est pourquoi, en comparant les patrimoines génétiques d'espèces diverses, on peut voir comment les gènes ont été modifiés au cours du temps, comment se sont différenciées les espèces et à quel degré elles sont apparentées.

I.1.8- Séquence nucléique : [Win99][09]

Les nucléotides sont mis bout à bout pour former une longue chaîne nommée *séquence nucléique* composant un brin d'ADN. La molécule d'ADN est formée de deux brins polynucléotidiques torsadés qui tournent dans des directions opposées : c'est une structure en double hélice, découverte par James Watson et Francis Crick en 1953.

Les deux brins d'ADN sont complémentaires : les bases adénine(A) et thymine(T) situées sur chaque brin se font toujours face, de même que les bases guanine(G) et cytosine(C). Donc en lisant la séquence des nucléotides sur un brin d'ADN, on peut déterminer la séquence de l'autre brin.

Notons que ces séquences sont orientées, c'est à dire la position des nucléotides sur les brins d'ADN à un sens biologique (exemple ATCG ≠ GTCA), car l'ADN n'est rien d'autre que le plan de fabrication de protéines dont la cellule a besoin pour se développer, survivre et accomplir son rôle dans l'organisme.

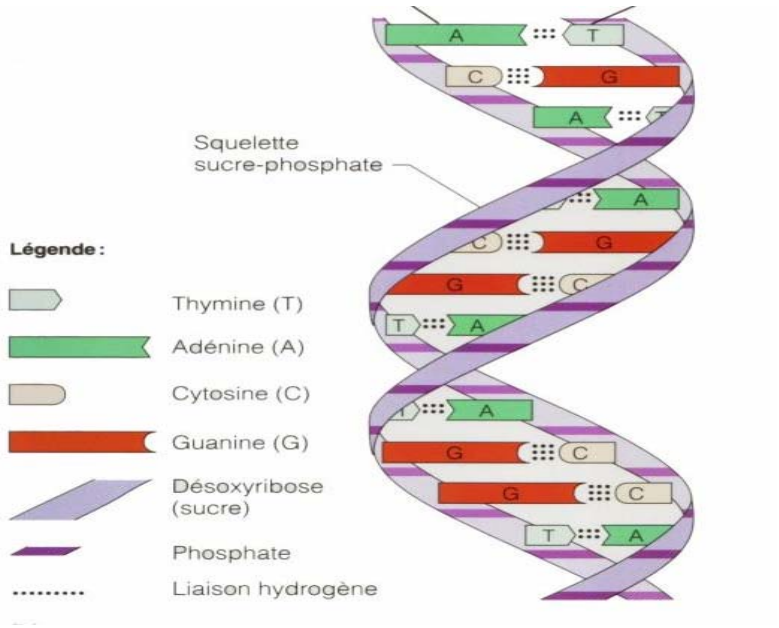


Figure 1.2 : la double hélice d'ADN.

I.1.9 Définition de la bioinformatique : [04, 06]

La bioinformatique est la discipline de l'analyse de l'information biologique, essentiellement sous la forme de séquences nucléiques et de structures de protéines. Elle utilise des méthodes de calcul spécifiques et de puissants ordinateurs, indispensables pour traiter la masse de données propre au domaine de la génétique.

Son but est :

- d'effectuer la synthèse des données disponibles à l'aide de modèles et de théories
- de formuler des prédictions (exemple : localiser et/ou prédire la fonction d'un gène) ;
- d'énoncer des hypothèses généralisatrices (exemple : comment les protéines se replient ou comment les espèces évoluent).

I.2 Notions de théorie des langages : [Cro01]

Définition 1 : Alphabets et Mot

Un ***alphabet*** est un ensemble fini non vide dont les éléments sont appelés des lettres.
Un ***mot*** sur un alphabet *A* est une suite finie d'éléments de *A*. La suite de zéro lettre est appelé le ***mot vide*** et notée ***ε***. On écrit un mot comme la simple juxtaposition des lettres qui le composent.
Exemple : Soit A={a,b}. ***ε, a, b*** et **baba** sont des mots sur A.
L'ensemble de tous les mots sur l'alphabet *A* est noté ***A*.***
$A^+ = A^* - \{ \varepsilon \}$.

Définition 2 : longueur d'un mot.

La ***longueur*** d'un mot x, notée $|x|$, est le nombre des lettres qui le composent.

Définition 3 : position d'un symbole.

On note ***x[i],*** pour $i=0,1,\ldots,|x|-1$, la lettre à l'indice i de x.

Définition 4 : identité de deux mots.

On dit que deux mots x et y sont identiques, noté $x=y$, si est seulement si :

- $|x|=|y|$
- $x[i]= y[i]$ pour $i=0,1,\ldots,|x|-1$.

Définition 5 : facteur, préfixe, suffixe.

Un mot x est un ***facteur*** d'un mot y s'il existe deux mots u et v tels que $y= uxv$.
Lorsque $u=\varepsilon$, x est un ***préfixe*** de y ; et lorsque $v=\varepsilon$, x est un ***suffixe*** de y.

Définition 6 : sous-mot.

Le mot x est un sous-mot de y si x est obtenu en supprimant des lettres de y.

Définition 7 : ordre lexicographique.

L'***ordre lexicographique***, noté $\leq$, est induit par un ordre sur les lettres des mots, noté de la même façon. Il est défini comme suit :
Pour $x, y \in A^*$, $x \leq y$ si et seulement si :

- soit x est préfixe de y,
- soit x et y se décomposent sous la forme $x = uav$ et $y = ubw$ avec $u,v,w \in A^*$, $a,b \in A$ et $a<b$

Exemple : $ababb < abba < abbaab$ en supposant $a<b$

Remarque : dans la suite de ce document, nous emploierons le terme « séquence » au lieu du terme « mot ».

I.3 Notions de théorie des ensembles : [07]

Définition 1 : Ensemble ordonné
Soit E un ensemble et R une relation sur E.
E est un ensemble ordonné si tous ses éléments sont comparables.

Définition 2 : Ordre partiel

Soit E un ensemble. Un ordre partiel sur E est une relation notée ≤ telle que pour tout $(x,y,z) \in E3$:

- $x \leq x$
- $(x \leq y \wedge y \leq x) \rightarrow x = y$
- $(x \leq y \wedge y \leq z) \rightarrow x \leq z$

Ces trois propriétés sont respectivement la réflexivité, la symétrie et la transitivité.
Un ensemble E muni d'un tel ordre est appelé «*ensemble partiellement ordonné*».

Exemple : Dans ℕ, la relation « *a* divise *b* », souvent notée par le symbole « ∣ « , est un ordre partiel.

Définition 3 : Ordre stricte

Soit E un ensemble. Une relation d'ordre stricte sur E est une relation binaire notée < vérifiant :

- $\forall$ x ∈ E, $x \not< x$ (irréflexivité)
- $\forall$ $(x,y,z) \in E^3$, $(x < y \wedge y < z) \rightarrow x < z$ (transitivité).

Définition 4 : Ordre total

Soit E un ensemble. Une relation d'ordre R sur E est totale si :
R est un ordre strict sur E.

$\forall (x,y) \in E^2$, soit xRy soit yRx.

Un ordre total est appelé ordre linéaire.

Exemple : Dans ℕ, la relation « < », est un ordre total.

Définition 5 : Chaîne et Antichaîne

Un ensemble totalement ordonné est aussi appelé une **chaîne.**
Un ensemble tel que $x \leq y \rightarrow x = y$ est appelé une **antichaîne**.

Définition 6 : Chaîne maximale

Une chaîne C est dite maximale si et seulement si quel que soit l'élément x, l'ensemble C∪{x} n'est pas une chaîne.
Une antichaîne C est dite maximale si et seulement si quel que soit l'élément x, l'ensemble C∪{x} n'est pas une antichaîne.
Lorsqu'une chaîne {x1 < x2 < … < xn} est maximale, alors ∀ i : xi < xi+1.

Définition 7 : Dual

Le dual d'un ensemble ordonné est le même ensemble mais muni de l'ordre ≤δ tel que x≤δy si et seulement si y ≤ x. Le dual d'un énoncé ψ est l'énoncé ψδ obtenu en remplaçant ≤ par ≥ et réciproquement.

Chapitre II

Le problème de la plus longue sous-séquences commune

Chapitre II : Le problème de la plus longue sous-séquence commune.

II.1 Problématique :

Etant donnée une séquence S de symboles définie sur un alphabet *A*, une sous-séquence de S est une séquence R qui peut être obtenue par la suppression ou pas d'un ou de plusieurs symboles. Le problème de la plus longue sous-séquence commune (PLCS-problem) à deux séquences données S = $s_1, s_2, \ldots, s_m$ et T = $t_1, t_2, \ldots, t_n$ (m≤n) consiste à trouver une troisième séquence R = $r_1, r_2, \ldots, r_l$ telle que :

- R est une sous-séquence de S ;
- R est une sous-séquence de T ;
- R est de longueur maximale.

Plusieurs solutions du PLCS-problem ont été proposées et classées suivant deux types d'approche : l'approche de programmation dynamique et l'approche de programmation non-dynamique.

II.2 Comparaison des séquences : [03]

Les programmes de comparaison des séquences ont pour but de repérer les endroits où se trouvent des régions identiques ou très proches entre deux séquences et d'en déduire celles qui sont significatives et qui correspondent à un sens biologique de celles qui sont observées par hasard.

En général les algorithmes de comparaison de séquences fonctionnent sur des segments de séquences (on parle de fenêtres, de motifs ou de mots) sur lesquels on regarde s'il existe ou pas une ressemblance significative. On distingue deux classes de ressemblances :

a- La ressemblance parfaite ou identité :

```
T  G  C  T  G  G  G  C  C  A
|  |     |  |  |     |
T  G  G  T  G  G  C  C  A  T
```

Segments identiques

Figure 2.1 : segments identiques

b- La ressemblance non parfaite que l'on qualifie de similitude :

```
T  G  G  G  C  C  A  C  C  T
      |  |  |  |  |     |  |
G  T  G  G  C  C  A  T  C  T
```

Segments similaires

Figure 2.2 : segments similaires

II.3 Définition de l'alignement des séquences : [02, 08]

Soient *S* et *T* deux mots définies sur l'alphabet ***A***= {*G*, *T*, *A*, *C*}. L'alignement de *S*, *T* est un couple de mots *S'*, *T'* sur l'alphabet $A \cup \{-\}$ où $- \notin A$

Le caractère d'espacement (gap), noté '–' satisfait les conditions suivantes :

- $|S'| = |T'|$;
- la suppression de '-' dans *S'* et dans *T'* donne *S* et *T* respectivement ;
- les caractères d'espacement '-' ne se retrouvent pas en même position dans *S'* et *T'*.

Soient a et b deux symboles de l'alphabet *A*. Un alignement peut être construit grâce à quatre opérations de base :

- l'appariement (*match*) : (a, a), un caractère de la première séquence est mis en regard d'un caractère identique dans la deuxième séquence ;
- la substitution : (a, b), un caractère de la première séquence est mis en regard d'un caractère différent dans la deuxième séquence ;
- l'insertion d'un gap dans S' : (-, b) ;
- l'insertion d'un gap dans T' : (a, -).

Exemple : soient S= *CGATTAG* et *T* = *GATCGA,* on peut aligner ces deux séquences comme suit :

$$S' = C\ G\ A\ T\ T\ -\ A\ G$$
$$T' = -\ G\ A\ T\ C\ G\ A\ -$$

II.3.1 Définition de la distance de Levenshtein : [Ber03]

Soit A* l'ensemble des mots constructibles à partir de l'alphabet A.

Soient S et T deux séquences définies sur A.

On dit qu'une fonction d : A* x A* ➔ $\mathbb{R}$ est une distance sur A* si les quatre propriétés suivantes sont respectées pour tout S, T $\in$ A* :

Positivité : $d(S, T) \geq 0$;
Séparation : $d(S, T) = 0 \Leftrightarrow S = T$;
Symétrie : $d(S, T) = d(T, S)$;
Inégalité triangulaire : $d(S, T) \leq d(S, R) + d(R, T)$. $\forall R \in A^*$.

La distance de Livenshtein, notée LD(S, T) est définie comme étant égale au nombre minimal d'opérations d'édition (insertion, suppression, substitution) nécessaires pour transformer la chaîne S afin d'obtenir la chaîne T.

Exemple : Soient S= « CHIENS » et T= « NICHE ».

Pour transformer la séquence S en T, on applique cinq opérations sur S :

- Suppression du S -> CHIEN
- Suppression du N -> CHIE
- Déplacement de I -> CIHE
- Déplacement de I -> ICHE
- Ajout du N -> NICHE

Donc LD(S, T) = 5.

II.3.2 Score d'alignement:

Soient deux séquences S = $s_1, s_2, \ldots, s_m$ et T = $t_1, t_2, \ldots, t_n$ définies sur l'alphabet A.

Soient a et b deux symboles de l'alphabet A, tel que $a \in S$ et $b \in T$.

Un score élémentaire indique la valeur qu'on veut attribuer à l'alignement du symbole a avec le symbole b à la position correspondante de l'autre séquence.

Un score global d'un alignement est égal à la somme des scores élémentaires calculés sur chacune des positions en vis-à-vis des deux séquences.

Le score global est alors égale à $\sum_{i=1}^{k} w(s'_i, t'_i)$ où $w(s'_i, t'_i)$ est le coût d'alignement de s'_i avec t'_i et k est la longueur commune à S' et T'.

Exemple : prenons l'exemple d'alignement précédent :

En supposant que : $w(a,b) = -1$ pour $a \neq b$, $w(a,a) = 5$ et $w(a, -) = w(-, b) = -2$. On trouve un score global égal à 13.

II.3.3 Matrice de score : [08]

Afin de déterminer le ou les meilleur(s) alignement(s) entre deux séquences S et T définies sur l'alphabet A= {A, C, G, T}, on attribue un coût à chaque opération d'alignement suivant une matrice de score élémentaire, notée w. Un exemple simple de matrice de score est la matrice identité :

w(s,t)	-	A	C	G	T
-	?	0	0	0	0
A	0	1	0	0	0
C	0	0	1	0	0
G	0	0	0	1	0
T	0	0	0	0	1

	appariement
	substitution
	Insertion d'un gap

Tableau 2.1 : Exemple de matrice de score.

Le score w(-,-) n'est pas défini puisque ce cas de figure n'est pas autorisé pour l'alignement par paires. On peut également modifier les coefficients afin de favoriser l'une des opérations d'édition. Par exemple si on utilise la matrice identité on cherche à favoriser les appariements et à minimiser les insertions de gaps.

On peut distinguer 2 types de matrices de score :

- Les matrices de score maximisantes : le score d'un appariement est supérieur aux autres scores : w(a,a) > w(a,b), w(a,-), w(-,a)
- Les matrices de score minimisantes : le score d'un appariement est inférieur aux autres scores : w(a,a) < w(a,b), w(a,-), w(-,a)

II.3.4 Le traitement des insertions et des délétions [01]

Il peut être nécessaire, pour conserver l'intégralité de l'information biologique, d'introduire des insertions ou des délétions de longueur variable à certaines positions des séquences. En fait, le traitement d'une délétion à l'intérieur d'une séquence est considéré comme une insertion dans la séquence lui faisant face. On utilise le terme « indel » (insertion-delétion) pour nommer ces opérations.

Les indels sont considérées comme des pénalités dans le calcul d'un score d'un alignement. Certains algorithmes attribuent une pénalité simple pour chaque insertion quelle que soit sa longueur ou bien pour chaque caractère inséré. D'autres algorithmes appliquent une pénalité fixe pour toute insertion plus une pénalité pour étendre l'insertion. Cette dernière pénalité peut être décrite par l'équation suivante : $P = x + yl$

où P est la pénalité pour une insertion de longueur l, x la pénalité fixe d'insertion et y la pénalité d'extension pour un élément.

Dans certains cas, le poids des pénalités peut être établi en fonction des endroits où elles se trouvent pour améliorer la sensibilité de la recherche. Par exemple, on peut définir des choix de pénalités à l'intérieur de régions protéiques ayant potentiellement une qualité physique ou chimique particulière.

II.3.5 Le Backtracking :

Le backtracking est une procédure qui permet de retrouver la plus longue sous-séquence commune en fonction de l'algorithme d'alignement utilisé.

Dans le cas des matrices d'alignement utilisées dans certains algorithmes, cette procédure consiste à remonter à partir de la cellule d'arrivée vers la(les) cellule(s) voisine(s). Le choix de la cellule voisine dépend du score recherché (maximal ou minimal). Cette procédure est réitérée jusqu'à arriver à la cellule initiale.

Si dans une cellule, on peut revenir vers plusieurs cellules voisines, alors il existe plusieurs chemins optimaux.

Exemple : soit la matrice d'alignement des deux séquences *S= AGGTCA* et *T=AGCCA*. Les valeurs de cette matrice sont obtenues par l'application d'une formule de récurrence de l'un des algorithmes d'alignement.

		A	**G**	**G**	**T**	**C**	**A**
	0	3	6	9	12	15	18
A	3	0	3	6	9	12	15
G	6	3	0	3	6	9	12
C	9	6	3	2	5	6	9
C	12	9	6	5	4	5	8
A	15	12	9	8	7	6	5

Tableau 2.2 : Exemple de backtracking.

Le backtracking se fait de la cellule d'arrivée (en bleu) vers la cellule initiale (en vert). Les cases grisées représentent le chemin parcourut et correspondent à l'alignement des deux séquences S et T.

```
Séquence S   A   G   G   T   C   A
             |       |       |   |
Séquence T   A   -   G   C   C   A
```

II.4 Algorithme de programmation dynamique :

II.4.1 Algorithme de Needleman et Wunsch : [Ber03] [01, 02, 05]

Nous nous intéressons ici à l'alignement global entre deux séquences. Les premiers à avoir utilisé la programmation dynamique pour l'alignement de séquences génétiques sont Needleman et Wunsch.

Soient deux séquences nucléiques S= AGGTCA de longueur n et R= AGCCA de longueur m. Pour former un alignement de S avec R, on construit progressivement une matrice dans laquelle la case de coordonnées (i, j) contient la distance entre le préfixe de longueur i de S et celui de longueur j de R. La case (n, m) de la matrice contient donc la distance entre S et R. On note S[i] le symbole à la position i dans S, et Si le préfixe de longueur i de S.

On considère les trois alignements suivants, pour obtenir un alignement entre Si et Rj :

- S_{i-1} et R_{j-1} et d'ajouter la mutation de S[i] en R[j]
- S_i et R_{j-1} et d'ajouter l'insertion de R[j]
- S_{i-1} et R_j et d'ajouter le délétion de S[i]

ensuite on choisi celui qui génère le coût minimal.

Etape 1 : *Construction de la matrice d'alignement.*

Si on note M la matrice d'alignement, C (i,j) le coût d'alignement de S[i] avec R[j],I le coût d'une insertion et D le coût d'une suppression, alors la formule de récurrence définie par Needleman et Wunsch est :

$$M(i, j) = \min \begin{cases} M(i-1, j-1) + C(i, j) \\ M(i, j-1) + I \\ M(i-1, j) + D \end{cases}$$

On se sert de cette formule de récurrence pour remplir chaque case (i,j) avec $i > 0$ et $j > 0$. La matrice d'alignement est de dimension n + 1 par m + 1.

L'initialisation de cette matrice se fait de la manière suivante :

- M (0, 0) = 0 ;
- M (i, 0) = i* coût (D) ; //coût d'alignement de S_{i-1} avec R_j
- M (0, j) = j* coût (I) ; //coût d'alignement de S_i avec R_{j-1}

On suppose que : Coût(i,j)=0 si S(i)=R(j), Coût(I) = Coût(D) = 3, Coût(i,j)=2 si S(i)≠R(i) qui correspond à une mutation. On a donc la matrice des coûts C :

	A	G	C	C	A
A	**0**	**2**	**2**	**2**	**0**
G	**2**	**0**	**2**	**2**	**2**
G	**2**	**0**	**2**	**2**	**2**
T	**2**	**2**	**2**	**2**	**2**
C	**2**	**2**	**0**	**0**	**2**
A	**0**	**2**	**2**	**2**	**0**

Tableau 2.3 : C : Matrice Coût

Alors la matrice d'alignement est initialisée comme suit :

S\R		A	G	C	C	A
	0	**3**	**6**	**9**	**12**	**15**
A	**3**					
G	**6**					
G	**9**					
T	**12**					
C	**15**					
A	**18**					

Tableau 2.4 : M : Matrice d'alignement initiale**.**

Le reste de la matrice est remplie à l'aide de la fonction de récurrence et de la matrice des coûts C.

On prend comme exemple le remplissage de la case M[1,1]= M(A,A), on a :

$$M(1,1) = \min \begin{cases} M(0,0) + C(1,1) = 0+0 = 0 \\ M(1,0) + 3 = 6 \\ M(0,1) + 3 = 6 \end{cases}$$

Donc on obtient M[1,1]= min (0,6,6) = 0.

On applique ce procédé sur le reste des cases, pour obtenir la matrice d'alignement représentée par la figure 2.4 :

S\R		A	G	C	C	A
	0	**3**	**6**	**9**	**12**	**15**
A	**3**	**0**	**3**	**6**	**9**	**12**
G	**6**	**3**	**0**	**3**	**6**	**9**
G	**9**	**6**	**3**	**2**	**5**	**8**
T	**12**	**9**	**6**	**5**	**4**	**7**
C	**15**	**12**	**9**	**6**	**5**	**6**
A	**18**	**15**	**12**	**9**	**8**	**5**

Tableau 2.5 : M : Matrice finale d'alignement.

Le score M[6,5]=5 indique que tout alignement optimal comportera 5 appariements (identité ou mutation).

Etape 2 : *Construction de la matrice des directions.*

Pour trouver l'alignement des deux séquences S et R, on fait le retour arrière à partir de la dernière case M[n+1,m+1]=M[6,5]. Ceci est possible en construisant la matrice des directions ***Dir*** obtenue par l'application de 3 mouvements autorisés :

a. le mouvement diagonal qui correspond au passage de la case (i,j) à la case (i-1,j-1). C'est un mouvement privilégie, car il correspond à une mutation ou une équivalence des deux symboles S[i] et R[j].
b. le mouvement horizontal qui correspond au passage de la case (i,j) à la case (i,j-1), qui correspond à l'alignement de R[j] avec un gap.
c. le mouvement vertical qui correspond au passage de la case (i,j) à la case (i-1,j), qui correspond à l'alignement de S[i] avec un gap.

La matrice des directions ***Dir*** est obtenue par les formules suivantes :

- Initialisation :

$$\begin{aligned} &\textbf{\textit{Dir}}[0,0] = x \\ &\textbf{\textit{Dir}}[i,0] = \leftarrow \quad \text{pour tout i de 1 à n} \\ &\textbf{\textit{Dir}}[0,j] = \uparrow \quad \text{pour tout j de 1 à m} \end{aligned}$$

- Calcul des directions :

$$\textbf{\textit{Dir}}[i,j] = \text{Union} \begin{cases} \nwarrow \text{si } M[I,j] = M[i-1,j-1] + C(S[i],R[j]) \\ \uparrow \text{si } M[I,j] = M[i-1,j] + C(S[i],-) \\ \leftarrow \text{si } M[I,j] = M[I,j-1] + C(-,R[j]) \end{cases}$$

S\R		A	G	C	C	A	*i*
	x	←	←	←	←	←	*0*
A	↑	↖	←	←	←	↖←	*1*
G	↑	↑	↖	←	←	←	*2*
G	↑	↑	↖↑	↖	↖←	↖←	*3*
T	↑	↑	↑	↖↑	↖	↖←	*4*
C	↑	↑	↑	↖	↖	↖	*5*
A	↑	↖↑	↑	↑	↖↑	↖	*6*
j	*0*	*1*	*2*	*3*	*4*	*5*	

Tableau 2.6 : ***Dir*** : Matrice des directions.

A partir de la matrice des directions on peut identifier 3 chemins différents (soit 3 alignements) figurés en rouge. Détaillons chacun de ces alignements :

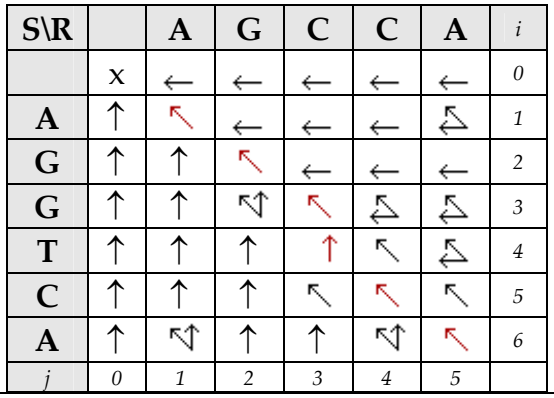
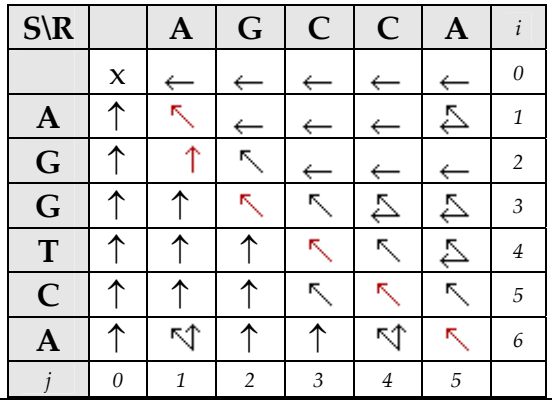
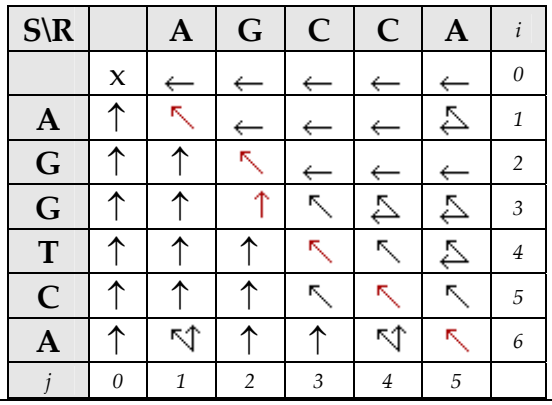
Chemin	Alignement
	**S = A G G T C A** **R = A G C - C A**
	**S = A G G T C A** **R = A - G C C A**
	**S = A G G T C A** **R = A G - C C A**

Tableau 2.7 : Les alignements correspondants aux chemins tracés.

Le coût de chaque alignement est égale à la valeur de la case M [6,5]=5 et c'est la distance d'édition des deux séquences. On peut vérifier ceci en appliquant un coût d'une mutation (M)=2, un coût d'une délétion (D)=3 et un coût d'équivalence (E)=0, sur le dernier alignement obtenu.

Séquence S :	**A**	**G**	**G**	**T**	**C**	**A**
Séquence R :	**A**	**G**	**-**	**C**	**C**	**A**
Coût :	0	0	3	2	0	0

Tableau 2.8 : Coût d'alignement des deux séquences S et R

Complexité de l'algorithme de Needleman & Wunsch :

Le remplissage de la matrice par l'algorithme Needleman&Wunsch s'effectue dans un temps d'ordre *O(nm)* où n et m sont les longueurs des deux séquences. La complexité spatiale est d'ordre (nm).

Lorsque les deux séquences sont de longueurs égales n=m, l'algorithme s'exécute en un temps quadratique, c'est-à-dire $O(n^2)$ du au fait qu'il parcourt les m*n cases de la matrice.

II.4.2 Algorithme de Smith & Waterman : [Smi81]

Dans ce qui suit nous nous intéressons à l'alignement local de deux séquences, et qui consiste à trouver le meilleur alignement d'un segment d'une séquence S avec une séquence R.

Une des méthodes d'alignement local les plus utilisées, basée sur la programmation dynamique, fut introduite par Smith et Waterman en 1981. La différence essentielle avec l'algorithme de Needleman et Wunsch(1970) est qu'on n'impose plus un alignement de toutes les lettres de chaque chaîne mais seulement d'un morceau de chaque chaîne, en plus n'importe quelle case de la matrice d'alignement peut être considérée comme point de départ pour le calcul des scores globaux. Le système de scores choisi possède des scores négatifs pour les mauvaises associations qui peuvent exister entre les éléments des séquences, et pour cette cause, on a ajouté le zéro dans la formule de récurrence, qui remplacera les sommes négatives des scores.

Formule de récurrence :

Soient deux séquences S et R de longueurs respectives n et m (avec n < m). La recherche d'un alignement local optimal entre S et R suivant une fonction de score c est obtenu par construction d'une matrice des scores optimaux d'alignement M[0..n,0..m] telle que :

- Initialisation : M[0,0] =0 ;
  M[i,0]=0 ;
  M[0,j]=0 ;

- Calcul du score optimal :

$$M[i,j] = \max \begin{cases} M[i-1,j-1] + c(x_i,y_j) \text{ // mutation(I,j)} \\ M[i-1,j] + c(x_i,-) \text{ // délétion}(x_i) \\ M[i,j-1] + c(-,y_j) \text{ // insertion}(y_j) \\ 0 \end{cases}$$

- où M[i,j] représente le score de l'alignement de S[1..i] avec R[1..j] et **C** est une matrice de score maximisante.

- Pour obtenir la position de la séquence R qui correspond au meilleur alignement possible de S avec R, on recherche la position j_{max} telle que la valeur $M[n,j_{max}]=\max(M[n,j])$ en partant de j=0.

  On peut alors créer un alignement local de S avec R en partant de $M[n,j_{max}]$ jusqu'à obtenir une valeur M[i,j]=0, donc on n'est pas obligé de commencer par la dernière case M[n,m] et remonter jusqu'à la première case M[0,0], comme c'est le cas dans l'algorithme de Needleman & Wunsch.

Fonctionnement :

Soient deux séquences nucléiques S={TGA} et R={TATGAACTA}

on suppose que :

c(x,x)=1 ;

c(x,y)=c(x,-)=c(-,y)=0 ; ce qui correspond respectivement à une mutation, délétion et une insertion.

La matrice des scores initiale, notée **C** obtenue est :

	T	**A**	**T**	**G**	**C**	**A**	**C**	**T**	**A**
T	1	0	1	0	0	0	0	1	0
G	0	0	0	1	0	0	0	0	0
A	0	1	0	0	0	1	0	0	1

Tableau 3.9 : W : matrice des scores initiale

On appliquant la formule de récurrence citée ci-dessus, on obtient la matrice des scores optimaux suivante, notée **M** :

		T	**A**	**T**	**G**	**C**	**A**	**C**	**T**	**A**
	0	0	0	0	0	0	0	0	0	0
T	0	1	1	1	1	1	1	1	1	1
G	0	1	1	1	2	2	2	2	2	2
A	0	1	2	2	2	2	3	3	3	3

Tableau 2.10 : M : matrice des scores optimaux**.**

On cherche la valeur j_{max} telle que M[3, j_{max}] = max M[3, j], en commençant de j=0.On trouve j_{max} = 6 et M[3, j_{max}]=3.

La matrice des directions ***Dir*** est obtenue par les formules suivantes :

- Initialisation :

$$\textit{Dir}[0,0] = x$$
$$\textit{Dir}[i,0] = x \text{ pour tout i de 1 à n}$$
$$\textit{Dir}[0,j] = x \text{ pour tout j de 1 à m}$$

- Calcul des directions :

$$\textit{Dir}[i,j] = \text{Union} \begin{cases} \nwarrow \text{ si } M[I,j] = M[i-1,j-1] + C(S[i],R[j]) \\ \uparrow \text{ si } M[I,j] = M[i-1,j] + C(S[i],-) \\ \leftarrow \text{ si } M[I,j] = M[I,j-1] + C(-,R[j]) \end{cases}$$

S\R	-	T	A	T	G	C	A	C	T	A	*i*
-	x	x	x	x	x	x	x	x	x	x	*0*
T	x	↖	←	↖	←	←	←	←	↖	←	*1*
G	x	↑	↖	↖	↖	←	←	←	←	←	*2*
A	x	↑	↖	←	↖	↖	↖	←	←	↖	*3*
j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	

Tableau 2.11 : ***Dir*** : Matrice des directions.

A partir de la matrice des directions on peut identifier 2 chemins différents (soit 2 alignements) figurés en rouge. Détaillons chacun de ces alignements :

S\R	-	T	A	T	G	C	A	C	T	A	*i*
-	x	x	x	x	x	x	x	x	x	x	*0*
T	x	↖	←	↖	←	←	←	←	↖	←	*1*
G	x	↑	↖	↖	↖	←	←	←	←	←	*2*
A	x	↑	↖	←	↖	↖	↖	←	←	↖	*3*
j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	

S= - - T G - A - - -
R= T A T G C A C T A

Tableau 2.12 : Alignement correspondant au premier chemin.

S\R	-	T	A	T	G	C	A	C	T	A	*i*
-	x	x	x	x	x	x	x	x	x	x	*0*
T	x	↖	←	←	←	←	←	←	⇦	←	*1*
G	x	↑	⇖	⇖	↖	←	←	←	←	←	*2*
A	x	↑	↖	←	↵	⇖	↖	←	←	⇦	*3*
j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	

S = **T - - G - A - - -**
R = **T A T G C A C T A**

Tableau 2.13 : Alignement correspondant au deuxième chemin.

Le coût de chaque alignement est égale à la valeur de la case M [3,6]=3 et c'est la distance d'édition des deux séquences. On peut vérifier ceci en appliquant les mêmes coûts utilisés pour la construction de la matrice, sur le dernier alignement obtenu.

Séquence S :	**T**	**-**	**-**	**G**	**-**	**A**	**-**	**-**	**-**
Séquence R :	**T**	**A**	**T**	**G**	**C**	**A**	**C**	**T**	**A**
Coût :	1	0	0	1	0	1	0	0	0

Tableau 2.14 : Coût d'alignement des deux séquences S et R

Complexité de l'algorithme de Smith & Waterman :

La complexité de cet algorithme pour deux séquences de taille n et m est de l'ordre *O(nm).* Ceci est dû au fait que le calcul d'une case (i,j) nécessite 4 comparaisons et 3 opérations arithmétiques, donc le remplissage de la matrice se fait en un temps *O(mn),* et également un temps d'ordre O(mn) pour rechercher la case de valeur maximale et pour trouver l'alignement optimal.

II.5 Algorithme de programmation non-dynamique :

II.5.1 Algorithme de Hunt & Szymanski : [Hun77, Apo87]

L'algorithme proposé par James W.Hunt et Thomas G.Szymanski en 1977, a pour objectif de réduire la complexité temporelle. L'algorithme proposé nécessite un temps d'exécution d'ordre *O((r+n) log n)*, où r est le nombre total des positions où deux séquences A et B se ressemblent, i.e les positions où A[i]=B[j].

Le principe de cet algorithme est de calculer, position par position, le rangement de tous les bons appariements dans chaque position de nucléotide de la première séquence.

La recherche se fait sur la chaîne la moins longue A, de longueur n. Elle consiste à calculer un tableau $T_{i,k}$ où i est le numéro du caractère courant dans x, et k est la longueur de la sous-séquence. Le caractère courant est comparé à chaque caractère de la chaîne B, en partant de la fin, puis plusieurs cas peuvent se produire :

1 / Le caractère égal se trouve à un seul endroit de B et le lien qui le lie avec celui de A ne croise pas un autre lien ; dans ce cas, on reporte la ligne précédente en ajoutant en $T_{i,k+1}$ l'indice de l'élément.

2/ Le caractère égal se trouve à un seul endroit de B et le lien qui le lie avec celui de A croise un autre lien ; dans ce cas on défait tous les liens précédents et on marque en $T_{i,k+1}$ l'indice de l'élément.

3/ Le caractère égal se trouve à plusieurs endroits de B ; dans ce cas, on reporte la ligne précédente en ajoutant en $T_{i,k+1}$ l'indice de l'élément, mais également les autres indices aux bons endroits en respectant la croissance des $T_{i,k}$.

Notations :

La structure de données utilisée dans cet algorithme est un tableau de « valeurs pointeurs » (en anglais Threshhold values) $T_{i,k}$ définit comme suit :
$T_{i,k}$ = le plus petit j tel que A[1 :i] et B[1 :j] contiennent une sous-séquence commune de longueur K.
Exemple : pour les deux séquences : A = abcbdda et B = badbabd :

$T_{5,1}$ =1 A abcbd B badbabd	$T_{5,2}$=3 A abcbd B badbabd
$T_{5,3}$=6 A abcbd B badbabd	$T_{5,4}$=7 A abcbd B badbabd

Donc chaque $T_{i,k}$ peut être considéré comme un pointeur qui permet de savoir combien a-t-on besoin de la séquence B pour produire la sous-séquence commune de longueur K avec les i premiers éléments de la séquence A.

On note que chaque case du tableau T est dans un ordre croissant, pour lequel les trois lemmes suivants ont été définis.

$$T_{i+1,k} = \begin{cases} \text{Le plus petit j tel que } A[i+1]=B[j] \\ \qquad\qquad \text{Et } T_{i,k-1} < j \leq T_{i,k} \\ T_{i,k} \text{ si un tel j n'existe pas.} \end{cases}$$

Pour chaque position i, on a besoin d'une liste des positions j correspondantes tel que A[i]=B[j], avec |A|=|B|=n. Ces listes, notée MATCHLIST, doivent être laissées dans l'ordre décroissant des j. Toutes les positions de la séquence A qui contiennent le même élément doivent utiliser la même MATCHLIST.

Pour les séquences A = abcbdda et B= badbabd, les listes désirées sont :

MATCHLIST[1] = <5,2>
MATCHLIST[2] = <6,4,1>
MATCHLIST[3] = <>
MATCHLIST[4] = MATCHLIST[2]
MATCHLIST[5] = <7,3>
MATCHLIST[6] = MATCHLIST[5]
MATCHLIST[7] = MATCHLIST[1]

L'algorithme : cet algorithme comporte 4 étapes

```
A[1 :n],B[1 :n] : tableaux d'éléments ;
THRESH[0 : n] :tableau d'entiers ;
MATCHLIST[1 :n] : tableau de listes ;
LINK    [1 :n] : tableau de pointeurs ;
PTR : pointeur ;
// étape 1 : construire les listes chaînées ;
pour I :=1 à n faire
        Initialiser  MATCHLIST[i] :=< j1,j2…,jp> tel que
                        j1 > j2 > … > jp  et A[i]=B[jq] pour 1 ≤ q ≤ p ;

// étape 2 : initialisation du tableau THRESH

THRESH [0] :=0 ;

pour I :=1 à n faire

THRESH[i]:= n+1;

LINK[0]:= null;

// étape 3 : calculer les valeurs successives de THRESH

pour i :=1 à n faire
  pour tout  j dans MATCHLIST[i] faire
     debut
      trouver k tel que  THRESH[k-1] < j ≤ THRESH[k] ;
      si j < THRESH[k] alors
         debut
              THRESH[k] := j;
              LINK[k]:= newnode (I,j,LINK[k-1])
        Fin ;
  Fin ;
```

// étape 4 : recouvrir la plus longue sous-chaîne commune dans l'ordre inverse

K := le plus large **k** tel que THRESH[k]≠ n+1 ;

PTR := LINK[k] ;

Tant que PTR ≠ null **faire**

début

imprimer la paire (i,j) pointée par **PTR** ;

avancer dans **PTR ;**

fin ;

On note que dans la 3ème étape de cet algorithme, la condition (***j < THRESH[k]***) est nécessaire pour ne pas se retrouver avec deux symboles de la séquence A qui pointent sur le même symbole de la séquence B.

Exemple d'application :

Soient les deux séquences *A = abcbdda* et *B = badbabd*.

Initialement, on a :

MATCHLIST[1] = <5,2>
MATCHLIST[2] = <6,4,1>
MATCHLIST[3] = <>
MATCHLIST[4] = MATCHLIST[2]
MATCHLIST[5] = <7,3>
MATCHLIST[6] = MATCHLIST[5]
MATCHLIST[7] = MATCHLIST[1]

LINK[0] :=Null

THRESH :	0	8	8	8	8	8	8	8

Pour i=1 et j=5 :

THRESH :	0	5	8	8	8	8	8	8

LINK[1] :=(1,5,LINK[0]);

Pour i=1 et j=2 :

THRESH :	0	2	8	8	8	8	8	8

LINK[1] :=(1,2,LINK[0]);

Pour i=2 et j=6 :

THRESH :	0	2	6	8	8	8	8	8

LINK[2] :=(2,6,LINK[1]);

Pour i=2 et j=4 :

THRESH :	0	2	4	8	8	8	8	8

LINK[2] :=(2,4,LINK[1]);

Pour i=2 et j=1 :

THRESH :	0	1	4	8	8	8	8	8

LINK[2] :=(2,1,LINK[1]);

Pour i=3 : on ne fait rien ;

Pour i=4 et j=6 :

THRESH :	0	1	4	6	8	8	8	8

LINK[3] :=(4,6,LINK[2]);

Pour i=4 et j=4 : on ne fait rien ;
Pour i=4 et j=1 : on ne fait rien ;

Pour i=5 et j=7 :

THRESH :	0	1	4	6	7	8	8	8

LINK[4] :=(5,7,LINK[3]);

Pour i=5 et j=3 :

THRESH :	0	1	3	6	7	8	8	8

LINK[2] :=(5,3,LINK[1]);

Pour i=6 et j=7 : on ne fait rien ;
Pour i=6 et j=3 : on ne fait rien ;

Pour i=7 et j=5 :

THRESH :	0	1	3	5	7	8	8	8

LINK[3] :=(7,5,LINK[2]);

Pour i=7 et j=2 :

THRESH :	0	1	2	5	7	8	8	8

LINK[2] :=(7,2,LINK[1]).

Le contenu final du tableau THRESH peut être représenté par la matrice suivante, en remplaçant les cases ayants la valeur « 8 » par un tiret « - » :

k / **i**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	0	**2**	-	-	-	-	-	-
2	0	1	**4**	-	-	-	-	-
3	0	1	4	-	-	-	-	-
4	0	1	4	**6**	-	-	-	-
5	0	1	3	6	**7**	-	-	-
6	0	1	3	6	7	-	-	-
7	0	1	2	6	7	-	-	-
j		**2**	**4**	**6**	**7**			

Ce tableau exprime bien le contenu du tableau THRESH, car la première colonne contient les indices i de la séquence A, la première ligne peut être utilisée pour connaître la longueur k des sous-séquences à chaque position de i ou j, et elle peut être utilisée pour retrouver cette PLCS à partir des valeurs de PTR. Les cases en bleu représentent les indices j retenus pour former le PLCS. Les valeurs en rouge représentent les indices des paires (i, j) formant la plus longue sous-séquence commune à A et B.

Le plus grand K est égal à 4, donc si on utilise PTR, et en ayant comme condition que les indices (i,j) des paires soient dans l'ordre décroissant, on obtient la plus longue sous séquence commune à A et B : (5, 7), (4, 6), (2, 4), (1, 2) qui correspond à l'alignement suivant :

Séquence A	-	a	-	b	c	b	d	d	a
		|		|		|	|		
Séquence B	b	a	d	b	a	b	d	-	-

Figure 2.3 : Alignement des séquences A et B.

Complexité de l'algorithme de Hunt et Szymanski :

Cet algorithme trouve et affiche la plus longue sous-séquence commune à deux séquences A et B en un temps d'ordre O((r+n) log n) et O(r+n) en espace. Ceci est dû au fait que :

- L'étape 1 de l'algorithme nécessite un temps d'ordre O(n log n) et un espace d'ordre O(n) ;
- L'étape 2 nécessite un temps d'ordre O(n) ;
- L'étape 3 nécessite un temps d'ordre O(n+ r log n) ;
- L'étape 4 nécessite un temps d'ordre O(n).

Cet algorithme est plus rapide dans les applications qui utilisent des séquences qui ont peu de positions en commun (matches).

II.5.2 Implémentation Diff :

Bien que le temps maximal d'exécution de l'algorithme de Hunt et Szymanski est d'ordre O(n^2 log n), il est rapide en pratique et surtout dans l'une de ses implémentations telle la commande **diff** d'UNIX.

Cette commande sert à comparer des fichiers. Elle compare les fichiers cibles ligne par ligne, séquentiellement. Dans certaines applications, telles que la comparaison de dictionnaires de mots, elle peut être utile pour filtrer les fichiers.

Diff *fichier-1 fichier-2* affiche en sortie les lignes qui différent des deux fichiers, avec des symboles indiquant à quel fichier appartient la ligne en question.

Chapitre III

Algorithme du Primal-Dual

Chapitre III : Algorithme du Primal-Dual :

Il y a deux approches générales au problème de la plus longue sous-séquence commune. L'approche de programmation dynamique prend un temps quadratique et un espace linéaire en fonction de la longueur de la séquence traitée, alors que l'approche non dynamique prend moins de temps mais plus d'espace. J.Y.Guo et F.K.Hwang [Guo05] ont proposé une nouvelle implémentation de l'approche non-dynamique qui semble tirer le meilleur profit en terme de temps et d'espace dans la comparaison des séquences d'ADN.

Cette nouvelle implémentation se base sur l'algorithme de Pevzner et Waterman [Pev93] pour ordonner les paires de nucléotides.

III.1 Algorithme du Primal-Dual de Pevzner et Waterman : [Guo05,Pev93]

III.1.1. Principe :

Soient deux séquences d'ADN $I=\{I_1, I_2,\ldots..,I_m\}$ et $J=\{J_1, J_2,\ldots..,J_n\}$ où $I_i,J_j \in$ {A,C,T,G}.

On définit $P=\{(i,j) : I_i = J_j\}$. On note aussi $\boldsymbol{P}=\{p_1,p_2,\ldots,p_l\}$ où chaque p_k est la paire (i_k, j_k) et l est le nombre des paires dans $\boldsymbol{P}$. On suppose que les deux séquences ont des longueurs du même ordre, soit $m=O(n)$. Ainsi $|\boldsymbol{P}|= O(n^2)$.

Pour appliquer l'algorithme de Pevzner et Waterman, on doit d'abord ordonner les paires selon trois ordres $\prec$, $\prec^*$ et $\sqsubset$.

Définition de l'ordre partiel $\prec$:

L'ordre partiel $\prec$ est défini par : $p_x \prec p_y$ Si $i_x < i_y$ et $j_x < j_y$.

Donc les paires ont la forme suivante :

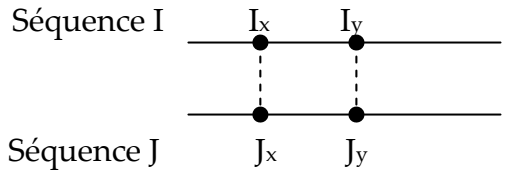


Figure 3.1 : ordre partiel $\prec$

Définition de l'ordre partiel $\prec^*$:

L'ordre partiel $\prec^*$ est défini par : $p_x \prec^* p_y$ Si $i_x \leq i_y$ et $j_x \geq j_y$
Donc les paires ont les formes suivantes, en supposant que le cas où $i_x = i_y$ et $j_x = j_y$ n'existe pas, car cela représente la paire elle-même.

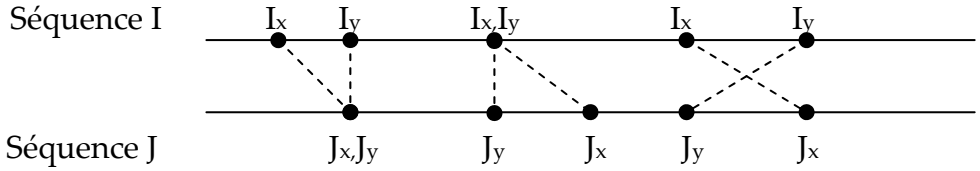


Figure 3.2 : ordre partiel $\prec^*$

Les ordres partiels $\prec$ et $\prec^*$ sont dites conjuguées si pour tout p_x, $p_y \in P$ la condition suivante est vérifiée :

p_x et p_y sont $\prec$ comparables $\Leftrightarrow$ p_x et p_y $\prec^*$ incomparables

Définition de l'ordre partiel $\sqsubset$:

L'ordre partiel $\sqsubset$ est défini par : $p_x \sqsubset p_y$ Si $p_x \prec p_y$ ou $p_x \prec^* p_y$
Donc les paires ont les quatre formes vues dans les définitions des ordres partiaux $\prec$ et $\prec^*$.

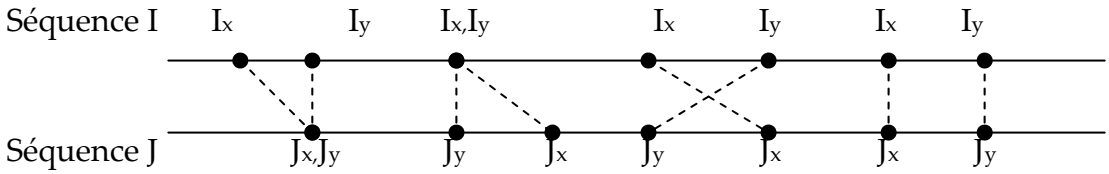


Figure 3.3 : ordre partiel $\sqsubset$

Lemme : Pevzner et Waterman ont prouvé que $\sqsubset$ est un ordre linéaire $p_1 \sqsubset p_2 \sqsubset \ldots \sqsubset p_l$, c'est-à-dire $\forall (p_x , p_y) \in \boldsymbol{P^2}$, p_x et p_y sont comparables et on a soit $p_x \sqsubset p_y$ soit $p_y \sqsubset p_x$.

Preuve : on prouve que $p_x \sqsubset p_y$ et $p_y \sqsubset p_z$ implique $p_x \sqsubset p_z$.
Si $p_x \sqsubset p_y$ et $p_y \sqsubset p_z$ alors l'une des conditions suivantes est vérifiée. :

1. $p_x \prec p_y$ et $p_y \prec p_z$,
2. $p_x \prec p_y$ et $p_y \prec^* p_z$,
3. $p_x \prec^* p_y$ et $p_y \prec p_z$,
4. $p_x \prec^* p_y$ et $p_y \prec^* p_z$.

- Dans le cas (1) : $p_x \prec p_y$ et $p_y \prec p_z$ implique $p_x \prec p_z$ et donc $p_x \sqsubset p_z$.
- Dans le cas (2) : $p_x \prec p_y$ et $p_y \prec^* p_z$ implique que ni $p_z \prec p_x$ ni $p_z \prec^* p_x$ n'est vérifiée. (Dans le premier cas $p_z \prec p_x$ et $p_x \prec p_y$ implique $p_z \prec p_y$ contredisant $p_y \prec^* p_z$. Dans le second cas $p_y \prec^* p_z$ et $p_z \prec^* p_x$ implique $p_y \prec^* p_x$ contredisant $p_x \prec p_y$). Donc $p_x \prec p_z$ ou $p_x \prec^* p_z$, implique $p_x \sqsubset p_z$.
- Les cas (3) et (4) sont, respectivement, identiques aux cas (2) et (1).

Donc on a démontré que la relation $\sqsubset$ est transitive. Suite à ces observations le lemme démontre que pour toutes paires p_x, p_y : soit $p_x \sqsubset p_y$ soit $p_y \sqsubset p_x$.

Puisqu'il existe l paires dans $\boldsymbol{P}$, et on ordonnera ces l paires par l'ordre partiel $\sqsubset$, et aux pire des cas on obtient n^2 paires, alors $|\sqsubset| = |\boldsymbol{P}| = l = O(n^2)$. La complexité évaluée pour cet algorithme est calculée pour le cas le plus défavorable (complexité au pire des cas).

Les paires $p_1, p_2, \ldots p_l$ sont regroupées dans des chaînes C_1, C_2,…tel que toutes les paires dans une même chaîne *Ci sont ordonnées par* $\prec^*$. Les paires d'une même chaîne sont toutes conjuguées.

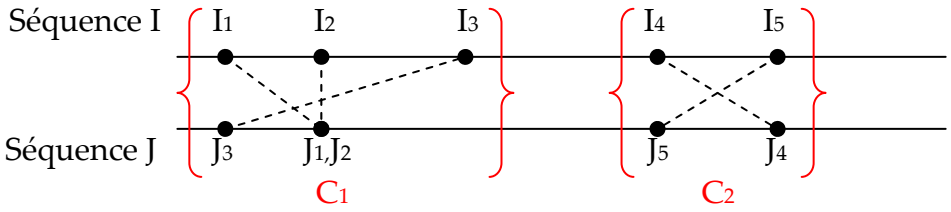


Figure 3.4 : Construction des chaînes

Le but de cet algorithme est de trouver un ensemble de paires ordonnées par la relation $\prec$ et de cardinalité maximale, ce qui correspond à trouver la plus longue sous-séquence commune à deux séquences. Ceci est réalisé par la construction des chaînes, où chaque chaîne contient un ensemble de paires conjuguées (odonnées par la relation $\prec^*$). Ensuite une paire par chaîne est retenue pour construire cette PLCS. Donc les paires retenues ne se croisent pas car elles sont ordonnées par la relation $\prec$.

- **Etape 1 :** construction des chaînes.

On suppose que $p_1, p_2, \ldots, p_u$ sont classées dans C_1, C_2,… C_v. Soit p_i^* l'élément maximal de chaque chaîne Ci, ainsi on obtient $p_1^*, p_2^*, \ldots, p_l^*$. Et soit k, $1 \le k \le v$, l'index minimal tel que $p_k^* \prec^* p_{u+1}$, p_{u+1} est l'élément en cours de traitement qu'on compare avec les p_k^*.

- Si $p_k^* \prec^* p_{u+1}$: donc on a trouvé un ordre $\prec^*$ à p_{u+1} dans cette chaîne, p_{u+1} est ajouté à C_k.

- Si un tel k n'existe pas, c'est à dire si aucune chaîne ne correspond au critère d'ordre partiel $\prec^*$ alors on crée une nouvelle chaîne C_{v+1} et on y met p_{u+1}.

Durant cette recherche, on calcule pour chaque paire la valeur de la fonction $b(p_{u+1})$ définie par :

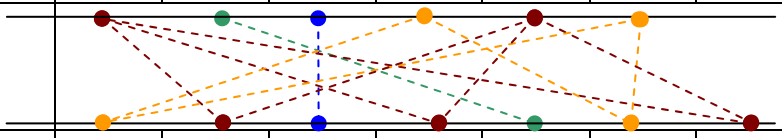
$$b(p_{u+1}) = \begin{cases} 03 & \text{si k=1,} \\ p^*_{k-1} & \text{si } 2 \le k \le v, \\ p^*_v & \text{si k n'existe pas.} \end{cases}$$

Il faut noter que *si* $b(p_{u+1}) \neq 0$ alors la paire $b(p_{u+1})$ n'est pas $\prec^*$ p_{u+1}, car $b(p_{u+1})$ appartient à une autre chaîne que celle de p_{u+1}, donc on sera sûr que durant le traçage de la plus longue sous-séquence commune(PLCS), on n'aura pas des paires qui se croisent au sein du PLCS.

Soient deux séquences d'ADN :

	I_1	I_2	I_3	I_4	I_5	I_6		J_1	J_2	J_3	J_4	J_5	J_6	J_7
***I*=**	T	G	C	A	T	A	***J*=**	A	T	C	T	G	A	T

La construction de l'ensemble ***P*** s'effectue par la comparaison des symboles de I et J :

	1	2	3	4	5	6	
Séquence I	T	G	C	A	T	A	
							
Séquence J	A	T	C	T	G	A	T
	1	2	3	4	5	6	7

P = { p_1 (1,2), p_2 (1,4), p_3 (1,7), p_4 (2,5), p_5 (3,3), p_6 (4,1), p_7 (4,6), p_8 (5,2), p_9 (5,4), p_{10} (5,7), p_{11} (6,1), p_{12} (6,6)}

L'application de l'ordre $\sqsubset$ donne la chaîne des paires suivante :

$\sqsubset$: p3, p2, p1, p4, p5, p7, p6, p10, p9, p8, p12, p11.

On obtient $p_3 \sqsubset p_2$ du fait que $p_3 \prec^* p_2$, et on obtient $p_1 \sqsubset p_4$ du fait que $p_1 \prec p_4$.

On classe le premier élément de la chaîne, $p_{u+1}=p_3$ dans C_1 (ici $U+1$ est l'indice de la première paire dans ***P***). Donc $p^*=p_3$ et puisque k=1 alors $b(p_3)=0$. Ce qui signifie qu'il sera probablement le premier élément de la sous-séquence commune.

Chaîne	C_1
paire	P_3
$b(p_{u+1})$	0

Pour $p_{u+1}=p_2$ ($p_u=p_3$), on le compare avec p^* de toutes les chaînes existantes, en commençant par la première chaîne (k=1). Dans ce cas il n'y a qu'une seule chaîne C_1, et comme $p_k^* = p_3$ et $p_3 \prec^* p_2$ alors p_2 sera mis aussi dans C_1 et son $b(p_{u+1})=0$.

$P_2 \prec^* p_1$ alors p_1 sera mis aussi dans C_1 et son $b(p_{u+1})=0$.

Chaîne	C_1		
paire	P_3	P_2	P_1
$b(p_{u+1})$	0	0	0

Quand on cherche à classer p_4, la chaîne C_1 contient les paires p_3, p_2, p_1. Pour savoir s'il faut mettre ou non p_4 dans C_1, il suffit de le comparer avec p_1 qui est le dernier élément inséré dans C_1. p_1= (1,2) et p_4= (2,5) ne sont pas ordonnés par $\prec^*$ donc on crée une nouvelle chaîne $C_{k+1}=C_2$ et on y met p_4. $b(p_4) = p_1$, car à cet instant $p_{k-1}^* = p_1$.

Chaîne	C_1			C_2
paire	P_3	P_2	P_1	P_4
$b(p_{u+1})$	0	0	0	P_1

Pour p_5= (3,3), on le compare avec l'élément maximal de C_1, c'est à dire p_1 : $p_1 \not\prec^* p_5$, puis avec p_4 où $p_4 \prec^* p_5$ alors on l'insère dans C_2 et $b(p_5)= p_1$.

Chaîne	C_1			C_2	
paire	P_3	P_2	P_1	P_4	P_5
$b(p_{u+1})$	0	0	0	P_1	P_1

Pour p_7= (4,6) : $p_1 \not\prec^* p_7$ et $p_5 \not\prec^* p_7$, donc on crée une nouvelle chaîne C_3 et on y met p_7. $b(p_7)= p_{k-1}^*=p_5$.

Classe	C_1			C_2		C_3
paire	P_3	P_2	P_1	P_4	P_5	P_7
$b(p_{u+1})$	0	0	0	P_1	P_1	P_5

Pour p_6= (4,1) : $p_1 \prec^* p_6$ alors on met p_6 dans C_1 et $b(p_6)= 0$.

Chaîne	C_1				C_2		C_3
paire	P_3	P_2	P_1	P_6	P_4	P_5	P_7
$b(p_{u+1})$	0	0	0	0	P_1	P_1	P_5

Pour p_{10}= (5,7) : $p_6 \not\prec^* p_{10}$, $p_5 \not\prec^* p_{10}$ et $p_7 \not\prec^* p_{10}$ donc on crée une nouvelle chaîne C_4 et on y met p_{10} dont $b(p_{10})=p_7$.

Chaîne	C_1				C_2		C_3	C_4
paire	P_3	P_2	P_1	P_6	P_4	P_5	P_7	P_{10}
$b(p_{u+1})$	0	0	0	0	P_1	P_1	P_5	P_7

Pour p_9= (5,4) : $p_6 \not\prec^* p_{10}$ et $p_5 \not\prec^* p_{10}$ mais $p_7 \prec^* p_9$ donc on met p_9 dans C_3. $b(p_9)=p_5$.

Chaîne	C_1				C_2		C_3		C_4
paire	P_3	P_2	P_1	P_6	P_4	P_5	P_7	P_9	P_{10}
$b(p_{u+1})$	0	0	0	0	P_1	P_1	P_5	P_5	P_7

Pour p_8= (5,2) : $p_6 \not\prec^* p_8$ mais $p_5 \prec^* p_8$, alors on met p_8 dans C_2. $b(p_8)=p_6$.

Chaîne	C_1				C_2			C_3		C_4
paire	P_3	P_2	P_1	P_6	P_4	P_5	P_8	P_7	P_9	P_{10}
$b(p_{u+1})$	0	0	0	0	P_1	P_1	P_6	P_5	P_5	P_7

Pour p_{12}= (6,6) : $p_6 \not\prec^* p_{12}$, $p_8 \not\prec^* p_{12}$ et $p_9 \not\prec^* p_{12}$ mais $p_{10} \prec^* p_{12}$, alors on met p_{12} dans C_4, $b(p_{12})=p_9$.

Chaîne	C_1				C_2			C_3		C_4	
paire	P_3	P_2	P_1	P_6	P_4	P_5	P_8	P_7	P_9	P_{10}	P_{12}
$b(p_{u+1})$	0	0	0	0	P_1	P_1	P_6	P_5	P_5	P_7	P_9

Enfin, Pour p_{11}=(6,1) : $p_6 \prec^* p_{11}$ donc on met p_{11} dans C_1, et $b(p_{11})=0$.

Le tableau suivant contient le résultat final de cette première étape :

chaîne	C_1					C_2			C_3		C_4	
paire	P_3	P_2	P_1	P_6	P_{11}	P_4	P_5	P_8	P_7	P_9	P_{10}	P_{12}
$b(p_{u+1})$	0	0	0	0	0	P_1	P_1	P_6	P_5	P_5	P_7	P_9

Tableau 3.1 : ordonnancement des paires.

- **Etape 2 :** Obtention de la plus longue sous séquence commune.

La plus longue sous séquence commune (PLCS) aux deux séquences *I* et *J* est obtenue en effectuant un retour arrière (backtracking) à partir des éléments de la dernière chaîne, notée C_L , en utilisant la fonction b. L représente la longueur de cette PLCS.

Si la dernière chaîne contient plus d'un élément, on aura plus d'une PLCS, car chaque paire de cette chaîne représente un début du chemin de backtracking.

Une fois cette PLCS identifiée, l'alignement correspondant peut être obtenu en remplissant entre les paires p_k et p_{k+1} les nucléotides inégalés par des gaps (-).

Dans notre exemple, si on commence par p_{12}, on obtient $p_{12} \succ p_9 \succ p_5 \succ p_1$ (voir le chemin en vert), et si on commence par p_{10}, on obtient $p_{10} \succ p_7 \succ p_5 \succ p_1$ (voir le chemin en rouge). Les deux séquences ont la même longueur L =|PLCS|=4.

Chaîne	C_1					C_2			C_3		C_4	
paire	P_3	P_2	P_1	P_6	P_{11}	P_4	P_5	P_8	P_7	P_9	P_{10}	P_{12}
b(p_{u+1})	0	0	0	0	0	P_1	P_1	P_6	P_5	P_5	P_7	P_9

Tableau 3.2 : extraction du PLCS par retour arrière (backtracking).

Si on utilise la première PLCS ($p_{12} \succ p_9 \succ p_5 \succ p_1$) qui correspondent aux paires {(6,6),(5,4),(3,3),(1,2)} = {(A,A),(T,T),(C,C),(T,T)} on obtient l'alignement suivant :

Séquence I	-	T	G	C	A	T	-	A	-
Séquence J	**A**	T	-	C	-	T	G	A	T
Paires du PLCS		p_1		p_5		p_9		p_{12}	

Figure 3.5 : Alignement du premier PLCS.

Si on utilise la deuxième PLCS ($p_{10} \succ p_7 \succ p_5 \succ p_1$) qui correspondent aux paires {(5,7),(4,6),(3,3),(1,2)} = {(T,T),(A,A),(C,C),(T,T)} on obtient l'alignement suivant :

Séquence I	-	T	G	C	-	-	A	T	A
Séquence J	**A**	T	-	C	T	G	A	T	-
Paires du PLCS		p_1		p_5			p_7	p_{10}	

Figure 3.6 : Alignement du deuxième PLCS.

III.1.2 Complexité de l'implémentation de Pevzner et Waterman :

Pevzner et Waterman ont évalué la complexité de cet algorithme comme suit :

Sachant que l= 12, n=7 et L= 4.

1- Cet algorithme nécessite O(n+l) temps et espace pour construire P car ils parcourent n positions dans la séquence J, et le nombre de paires que peut contenir P est égale à l (ici l=12)

2- Cet algorithme nécessite O(l log l) temps pour ordonner $\boldsymbol{P}$ suivant l'ordre $\sqsubset$, car c'est un tri linéaire d'une séquence d'éléments qui représentent des paires.

3- Il nécessite aussi O(lL) temps parce qu'il traite les l éléments de P, et compare chaque élément avec les p* des L chaînes existantes.

4- Il nécessite O(l+L) espace pour construire C_1,C_2,… C_L. Car les éléments de P nécessitent l emplacements mémoire et les chaînes obtenues nécessitent L emplacements mémoire.

III.2 Algorithme de Hwang et Guo : [Guo05]

L'implémentation de l'algorithme du Primal-Dual dérivé de Pevzner et Waterman nécessite $O(l+ Ln)$ en temps et autant en espace mémoire. L'algorithme de Hwang et Guo œuvre pour réduire cette complexité et la rendre d'ordre $O(nL)$ en temps et $O(n)$ en espace. Cet algorithme est fondé sur les concepts de base de l'algorithme du primal-dual mais l'implémente d'une manière à gagner du temps en ne traitant qu'un nombre limité des paires.

III.2.1 Principe de l'algorithme :

Cet algorithme diffère de l'algorithme de Pevzner & Waterman par le fait qu'il permet de gagner du temps en ne parcourant qu'un nombre limité de paires, car l'algorithme de Pevzner et Waterman analyse tous les éléments de l'ensemble $\boldsymbol{P}$ pour trouver la paire suivante.

Dans cet algorithme, on n'a pas besoin de constituer un ensemble de paires et de les ordonner, car la construction de ces paires et des chaînes qui les regroupent se fait d'une manière progressive, tout en optimisant les opérations de comparaisons à l'aide du tableau des positions des nucléotides, noté Y.

Pour sélectionner l'ensemble des paires nécessaires à la recherche du PLCS, on construit un tableau X, à l'aide du tableau Y. Le contenu final de ce tableau constitue les chaînes qui contiennent chacune une seule paire, non-conjuquée avec la paire d'une autre chaîne. Ces paires représentent la plus longue sous-séquence commune, équivalente à un éventuel alignement des deux séquences. Cette PLCS est obtenue à l'aide de la fonction b qui permet de faire le retour arrière, et qui est calculée durant la construction du tableau X.

1ère étape : *construction du tableau X :*

On construit un tableau X avec 5 lignes marquées par j, A, C, G, T et n+1 colonnes marquées par n, n-1,….,1,0 (les index de la séquence J). Chaque colonne représente les prochaines positions des symboles dans la séquence J à partir de la position j. La colonne n est vide, parce qu'il n'y pas de prochaine position à partir de la dernière position j = n. Si un symbole N dans J est à la position n, alors on rempli la colonne X[N, n-1] par la valeur de n, ainsi que les colonnes avant n-1.

Par exemple, pour J = ATCTGAT

Indice j : 1 2 3 4 5 6 7

- On a l'index de A = 1, alors on remplit la X[A, 0] := 1 ;
- Pour l'index de T = 2 : X[T, 1] = X[T, 0] := 2 ;
- Pour l'index de C=3 : X[C, 2] = X[C, 1] = X[C, 0] := 3 ;
- …
- Pour l'index de T=7 : X[T, 6] = X[T, 5] = X[T, 4] := 7 ;

Et on obtient le tableau X suivant :

	J_1	J_2	J_3	J_4	J_5	J_6	J_7
***J*=**	A	T	C	T	G	A	T

j	7	6	5	4	3	2	1	0
A	-	-	6	6	6	6	6	1
C	-	-	-	-	-	3	3	3
G	-	-	-	5	5	5	5	5
T	-	7	7	7	4	4	2	2

Tableau 3.3 : Contenu du tableau X

La première colonne (j=0) permet d'avoir la première position de chaque nucléotide, puisqu'on est à la position j=0. On remarque aussi que les valeurs dans chaque ligne sont dans un ordre croissant, c'est dû au fait que les indices j commencent du début de la séquence J et qu'à chaque position d'indice j on a la prochaine position de chaque nucléotide.

2ème étape : *construction du tableau Y :*

Dans cette étape, on construit le tableau Y avec L+1 colonnes (L est inconnu au début) marquées par C_0, C_1, .., C_L, et 6 lignes marquées par j, i, A, C, G, T ; tout en calculant la valeur d'une fonction, notée ***b,*** nécessaire pour la construction du PLCS en affectant à chaque paire d'une chaîne Ci une paire de la chaîne qui la précède Ci-1, ces deux paires sont $\prec$ comparables.

Durant la construction de ce tableau, on cherche à établir une liste de paires d'indices (i, j) d'un même symbole (identité), tout en essayant de maximiser la cardinalité de cette liste, en exploitant le maximum de positions sur la deuxième séquence, à fin d'obtenir le(s) PLCS.

Pour maximiser le nombre des paires de symboles identiques, on procède à une comparaison des positions pour trouver une éventuelle position vide dans la partie déjà parcourue de la séquence J. Cette comparaison est effectuée à chaque création d'une nouvelle colonne, ou lorsqu'on n'a pas de prochaine position.

Au début, seules les entrées de la colonne C_0 sont remplies par 0, 0, A(0), C(0), G(0), T(0), c'est à dire la dernière colonne du tableau X.

Dans la séquence I de l'exemple suivant, la première position i=1 est du symbole T. Dans la ligne T du tableau X on trouve T(0)=2, alors on remplit la colonne C_1 avec les entrées 2, 1, A(T(0)), C(T(0)), G(T(0)), T(T(0)), et on affecte à b(1, T(0)) la valeur (0.0). Donc on obtient la première paire (T, T) d'indices (1,2) et les prochaines positions de chaque symbole dans la séquence J.

	I1	*I2*	*I3*	*I4*	*I5*	*I6*
***I*=**	T	G	C	A	T	A

	C_0	C_1
j	0	2
i	0	1
A	1	6
C	3	3
G	5	5
T	2	4
b(1,2)=(0,0)		

Tableau 3.4 : Tableau Y au début de la construction

		1	2	3	4	5	6	
I	=	**T**	**G**	**C**	**A**	**T**	**A**	
J	=	**A**	**T**	**C**	**T**	**G**	**A**	**T**
		1	2	3	4	5	6	7

Soit la ligne N qui a par exemple des entrées $n_0 \leq n_1 \leq \ldots \leq n_k$ pour k≤L. Pour chaque $n_{i\ (i<k)}$, dans l'ordre du plus grand au plus petit, on procède comme suit :

Soit j_w la valeur de j dans la colonne C_w, 0≤w≤k. On compare n_k avec $j_{w(k)}$, $j_{w(k-1)}$,…. Jusqu'à la première colonne $C_{w(k)}$ tel que $j_{w(k)} < n_k$. On remplit la colonne suivante $C_{w(k)+1}$ (ou on remplace ses entrées) avec n_k, y, A(n_k), C(n_k), G (n_k), T(n_k). Et b(y, n_k) = (i, j) où (i, j) est de $C_{w(k)}$.

L'idée est de comparer les valeurs dans la ligne de chaque symbole, notées n_w avec les valeurs de la ligne de j notées j_w, (avec$0 \leq w \leq k$) et si une valeur est strictement inférieur à j_w, alors on remplace les valeurs de la colonne suivante par la nouvelle valeur de j, la position actuelle de i, ainsi que les prochaines positions de chaque symbole sur la séquence J. On arrête cette comparaison lorsqu'on arrive à un $j_w < n_k$.

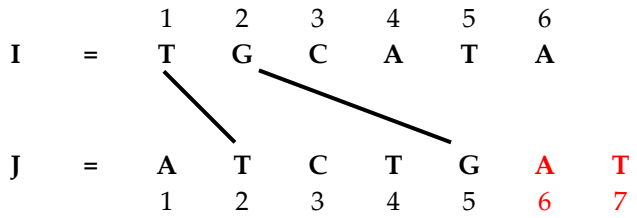
Cette procédure est démontrée par l'exemple déjà traité ci-dessus :

index	*1*	*2*	*3*	*4*	*5*	*6*		*index*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
I :	T	G	C	A	T	A		***J :***	A	T	C	T	G	A	T

Jusque là, on a obtenu le tableau Y contenant les deux colonnes C_0 et C_1, à partir de cette dernière colonne et du fait que l'index I est sur la deuxième position, qui correspond à G.

A partir de la colonne C1, on sait que la prochaine position du symbole G est 5, donc on crée une nouvelle colonne C_2, qu'on remplit par les entrées G(2), 2, A(G(2)), C(G(2)), G(G(2)), T(G(2)) ; avec G(2)= 5 dans la table X, b(2,5)=(1,2). Et comme Y[G, C0]=5 $\not<$ Y[J, C2]=5, alors on ne fait rien.

	C_0	C_1	C_2
j	0	2	5
i	0	1	2
A	1	6	6
C	3	3	-
G	5	5	-
T	2	4	7
b(1,2)=(0,0), b(2,5)=(1,2)			

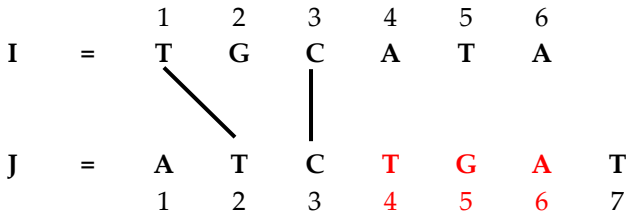


Pour i =3 où N = 'C' : C(5) = '-' donc on n'a pas de prochaine position du symbole C sur la séquence J, on ne fait rien.

On doit appliquer, dans ce cas, la comparaison des positions de C, notées Y[C,C_w] avec les valeurs de j, pour w<k. Comme Y[C,C_1]=3 < 5, alors on remplace les entrées de la colonne C_2 par 3, 3, A(C(2)), C(C(2)), G(C(2)), T(C(2)), et on calcule b(3,C(2))=b(3,3)=(1,2), après avoir recopié le tableau Y.

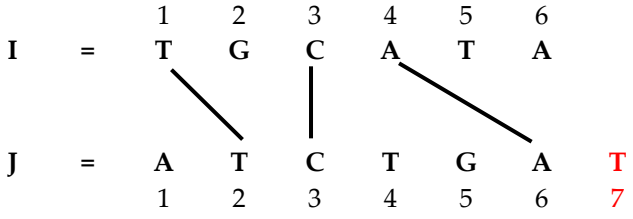
	C_0	C_1	C_2
j	0	2	3
i	0	1	3
A	1	6	6
C	3	3	-
G	5	5	5
T	2	4	4
b(1,2)=(0,0) ;b(2,5)=(1,2) ;b(3,3)=(1,2)			

Ensuite Y[C, C_0]=3 $\not<$ Y[j,C_2]=3, donc on ne fait rien.



-Pour i=4 : I(i)='A', A(3)=6 alors par le biais du Tableau X, on remplit la nouvelle colonne C_3 par les entrées A(6)=6, i=4, A(6), C(6), G(6), T(6). B(4,6)=(3,3).

	C_0	C_1	C_2	C_3
j	0	2	3	6
i	0	1	3	4
A	1	6	6	-
C	3	3	-	-
G	5	5	5	-
T	2	4	4	7
b(1,2)=(0,0)) ; b(2,5)=(1,2) ; b(3,3)=(1,2) ; b(4,6)=(3,3)				

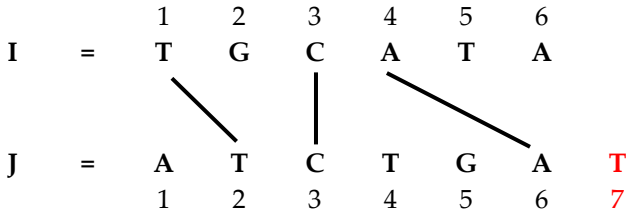


Y[A,C_2]=6 $\not<$ Y[j,C_3]=6, alors on ne fait rien.

Y[A,C_0]=1 < Y[j,C_1]=2, donc on remplace les entrées de la colonne C_1 par 1,4, A(1), C(1), G(1), T(1), bien sûr après avoir recopié le tableau Y, et on calcule b(4,1)=(0,0).

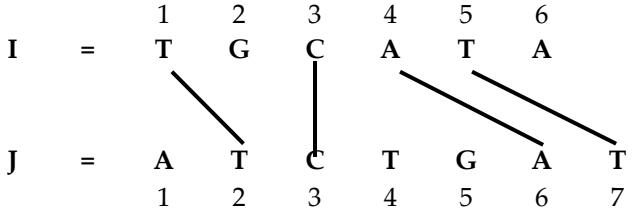
Le résultat de ces transformations est présenté dans le tableau Y suivant :

	C_0	C_1	C_2	C_3
j	0	1	3	6
i	0	4	3	4
A	1	6	6	-
C	3	3	-	-
G	5	5	5	-
T	2	2	4	7
b(1,2)=(0,0) ; b(2,5)=(1,2) ; b(3,3)=(1,2) ; b(4,6)=(3,3)				



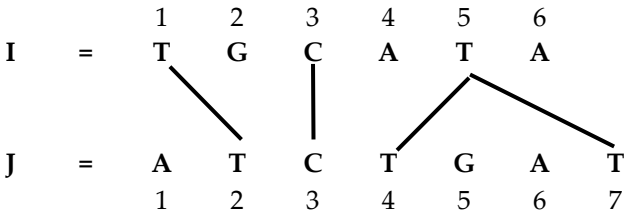
- En arrivant à i=5, qui correspond au nucléotide T : T(6)=7, on remplit une nouvelle colonne C_4 par les valeurs 7, 5, -, -, -, -. B(5,7)=(4,6).

	C_0	C_1	C_2	C_3	C_4
j	0	1	3	6	7
i	0	4	3	4	5
A	1	6	6	-	-
C	3	3	-	-	-
G	5	5	5	-	-
T	2	2	4	7	-
b(1,2)=(0,0) ; b(2,5)=(1,2) ; b(3,3)=(1,2) ; b(4,6)=(3,3) ; b(5,7)=(4,6)					



Y[T,C_2]=4 < 6 : on remplace les entrées de la colonne C_3 par : 4, 5, 6, -, 5, 7 ; après avoir recopié le tableau Y. b(5,4) = (3,3).

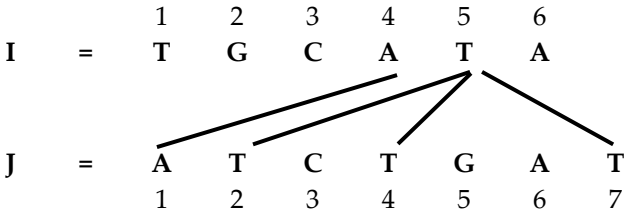
	C_0	C_1	C_2	C_3	C_4
j	0	1	3	4	7
i	0	4	3	5	5
A	1	6	6	6	-
C	3	3	-	-	-
G	5	5	5	5	-
T	2	2	4	7	-
b(1,2)=(0,0) ; b(2,5)=(1,2) ; b(3,3)=(1,2) ; b(4,6)=(3,3) ; b(5,7)=(4,6) ; b(5,4)=(3,3)					



Y[T,C_1]=2 < 4 : on remplace les entrées de la colonne C_2 par : 2, 5, 6, 3, 5,4. b(5,2)= (4,1).

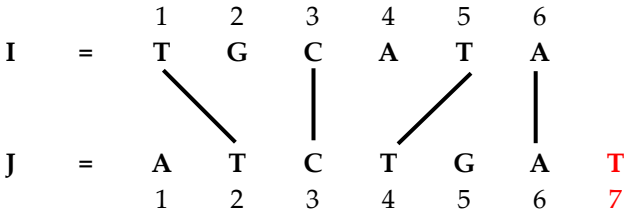
Et comme Y[T,C_0]=2 $\not<$ 2, alors on ne fait rien.

	C_0	C_1	C_2	C_3	C_4
j	0	1	2	4	7
i	0	4	5	5	5
A	1	6	6	6	-
C	3	3	3	-	-
G	5	5	5	5	-
T	2	2	4	7	-
b(1,2)=(0,0) ; b(2,5)=(1,2) ;b(3,3)=(1,2) ; b(4,6)=(3,3) ; b(5,7)=(4,6) ; b(5,4)=(3,3)					



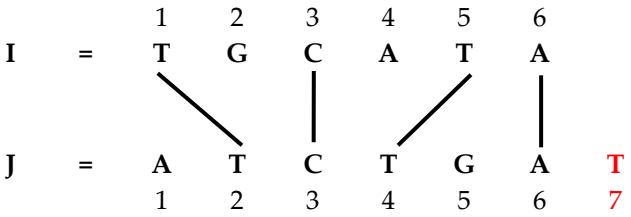
- Pour i=6, on a I(6)= 'A' et A(7)='-' alors on ne fait rien.
- Y[A,C_3]=6<7 : on remplace les entrées de la colonne C_4 par : 6, 6, -, -, -, 7. b(6,6)=(5,4).

	C_0	C_1	C_2	C_3	C_4
j	0	1	2	4	6
i	0	4	5	5	6
A	1	6	6	6	-
C	3	3	3	-	-
G	5	5	5	5	-
T	2	2	4	7	7
b(1,2)=(0,0) ; ;b(2,5)=(1,2) ; b(3,3)=(1,2) ; b(4,6)=(3,3) ; b(5,7)=(4,6) ; b(5,4)=(3,3) ; b(6,6)=(5,4) ; b(6,6)=(5,4)					



- Y[A,C_2]6 ≮ 6, alors on ne fait rien.
- Y[A,C_1]=6 ≮ 4, alors on ne fait rien.
- Y[A,C_0]=1<2 : on remplace les entrées de la colonne C_1 par : 1, 6, 6, 3, 5, 2. b(6,1)=(0,0).

	C_0	C_1	C_2	C_3	C_4
j	0	1	2	4	6
i	0	6	5	5	6
A	1	6	6	6	-
C	3	3	3	-	-
G	5	5	5	5	-
T	2	2	4	7	7
b(1,2)=(0,0) ; b(2,5)=(1,2) ;b(3,3)=(1,2) ; b(4,6)=(3,3) ; b(5,7)=(4,6) ; b(5,4)=(3,3) ; b(6,6)=(5,4) ; b(6,1)=(0,0)					

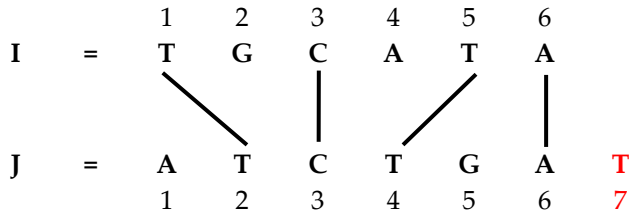


Le tableau suivant correspond aux différentes transformations effectuées pour construire le tableau Y :

	C_0	C_1	C_2		C_0	C_1	C_2	C_3		C_0	C_1	C_2	C_3	C_4
J	0	2	5		0	2	3	6		0	1	3	6	7
I	0	1	2		0	1	3	4		0	4	3	4	5
A	1	6	6		1	6	6	-		1	6	6	-	-
C	3	3	-		3	3	-	-		3	3	-	-	-
G	5	5	-		5	5	5	-		5	5	5	-	-
T	2	4	7		2	4	4	7		2	2	4	7	-
b(1,2)=(0,0) ; b(3,3)=(1,2) ; b(5,7)=(4,6) ; b(2,5)=(1,2), b(4,6)=(3,3), b(4,1)=(0,0)														

	C_0	C_1	C_2	C_3	C_4		C_0	C_1	C_2	C_3	C_4
j	0	1	3	4	7		0	1	2	4	6
i	0	4	3	5	5		0	6	5	5	6
A	1	6	6	6	-		1	6	6	6	-
C	3	3	3	-	-		3	3	3	-	-
G	5	5	5	5	-		5	5	5	5	-
T	2	2	4	7	-		2	2	4	7	7
b(5,4)=(3,3) ; b(6,6)=(5,4) ; b(5,2)=(4,1) ; b(6,1)=0											

Tableau 3.5 : les différents états du Tableau Y



Le tableau suivant montre la différence en nombre de paires traitées entre ces deux implémentations :

Implémentation / Classe	Pevzner & Waterman	Hwang & Guo
C1	(1,2), (1,4), (1,7), (4,1), (6,1)	(1,2), (4,1), (6,1)
C2	(2,5), (3,3), (5,2)	(2,5), (3,3), (5,2)
C3	(4,6), (5,4)	(4,6), (5,4)
C4	(5,7), (6,6)	(5,7), (6,6)

Donc en faisant un petit calcul, nous saurons qu'on a gagné 17% (2 paires non traitées) du temps de traitement pour un ensemble de 12 paires.

On remarque que les valeurs des paires $(Y[i,C_k], Y[j,C_k])$ de chaque C_k $(0 \le k \le L)$ correspondent aux paires p^* vues dans la présentation de l'algorithme primal-dual de Pevzner et Waterman [Pev93].

Maintenant, la(ou les) plus longue sous-séquence de longueur L est obtenu en partant d'une paire de la colonne C_L et à l'aide de la fonction de retour arrière b.

Dans l'exemple ci-dessus, si on prend la paire (6,6) dans la colonne C_4, et à l'aide de b(6,6)=(5,4), b(5,4) = (3, 3), b(3,3) = (1,2), on obtient le PLCS : (I_1, J_2), (I_3, J_3), (I_5, J_4), (I_6,J_6).

		1	2	3	4	5		6	
Séquence I =	**-**	**T**	**G**	**C**	**A**	**T**	**-**	**A**	**-**
		|		|		|		|	
Séquence J =	**A**	**T**	**-**	**C**	**-**	**T**	**G**	**A**	**T**
	1	2		3		4	5	6	7

Figure 3.7 : Premier alignement des séquences I et J.

Si on commence à partir de la paire (5,7), alors on obtiendra (I_1,J_2) , (I_3,J_3), (I_4,J_6), (I_5,J_7).

		1	2	3			4	5	6
Séquence I =	**-**	**T**	**G**	**C**	**-**	**-**	**A**	**T**	**A**
		|		|			|	|	
Séquence J =	**A**	**T**	**-**	**C**	**T**	**G**	**A**	**T**	**-**
	1	2		3	4	5	6	7	

Figure 3.8 : Deuxième alignement des séquences I et J.

On démontre par la suite que ce procédé est en effet une implémentation de l'algorithme primal-dual. On traite les paires dans ***P*** dans l'ordre lexicographique de (i,j). Les paires ayant le même i, appelées *i^{eme}-batch*, sont traitées consécutivement. On peut observer ça durant l'exécution de cet algorithme, car pour chaque position d'un symbole de la séquence I, on cherchait les positions des symboles équivalents dans la séquence J. Par exemple pour le symbole T dans la séquence I, on a obtenu trois paires : (5,2), (5,4), (5,7) formant le $5^{ème}$ batch.

On suppose que le traitement s'effectue sur un i^{eme}-batch, et que C_1,…, C_k ne sont pas vides. Soit $(i_1.j_1)$, …, (i_k,j_k) les paires maximales dans C_1,…,C_k, respectivement. Alors $j_1 < j_2 < \ldots < j_k$. Ceci revient à démontrer que les paires dans les C_1,…,C_k ne sont pas conjuguées.

Il suffit de prouver que $j_w < j_{w+1}$:
On note p_w la paire ayant pour indices (i_w, j_w).

- Si la paire p_w est traitée avant la paire p_{w+1}, alors $i_w \leq i_{w+1}$ et $j_w < j_{w+1}$, ou p_{w+1} doit être assignée à C_w. Ça correspond au cas $p_k^* \prec^* p_{u+1}$ vue dans l'algorithme de Pevzner et Waterman.
  Exemple : $p_4 \prec^* p_5$ => assigner p_5 à C_2.

- Si p_w est traitée après, et p'_w a été la paire maximale de C_w quand la paire p_{w+1} a été traitée, alors $i_w \leq i'_w \leq i_{w+1}$ et $j_w \leq j'_w < j_{w+1}$. Car si $j'_w \leq j_{w+1}$ alors on aurait assigné l'une de ces deux paires à la même colonne.

  Ça correspond à 2 cas de figures :
  - Soit $p_w \prec^* p'_w$ alors on effectue un remplacement des entrées de la colonne Cw en lui assignant (i_w, j_w) ;
  - Soit $p_w \prec p'_w$ donc p'_w restera l'élément maximal de la colonne Cw.

On note que toute paire p' traitée avant le i^{eme}-batch, a i'<i. Par conséquent, la $i^{ème}$ paire peut être soit $*\succ$ p', soit incomparable, mais pas plus petite. Autrement dit, (i,j) $*\succ$ (i_h, j_h) si et seulement si $j \leq j_h$. Ainsi un la i^{eme} paire (i,j) est mise dans C_h ssi $j_{h-1} < j \leq j_h$. Car à cet instant-là (i,j) devient la paire maximale de la chaîne C_h. Par contre si $j > j_h$, alors (i,j) est la première paire d'une nouvelle chaîne C_{k+l}. Ainsi les paires dans un i-batch sont divisées en plusieurs intervalles où les paires d'un intervalle vont à la même C_h. On note aussi que les $i^{èmes}$-paires sont toujours comparables par la relation d'ordre $\prec^*$, puisque $j_1^* < j_2^* < \ldots < j_g^*$ donc $(i, j_1^*) *\succ (i, j_2^*) *\succ \ldots *\succ (i, j_g^*)$.

Ainsi on doit mettre seulement une paire (i, j) de chaque intervalle h dans C_h où j est minimal parmi toutes les i-paires de cet intervalle pour qu'elle soit la paire maximale. Il est facile de vérifier que la paire (i,j) dans la colonne C_h du tableau Y est en effet la paire maximale (i_h, j_h) de la chaîne C_h. Ainsi l'entrée Y [N, C_h] donne l'index minimal $x > j_h$ du symbole de type N(la prochaine position du symbole de type N) à partir de la position j actuelle. Par conséquent, si N est le prochain symbole à traiter, alors toutes les valeurs j des paires maximales dans C_1, .., C_k ; (C_0 donne le j minimal) sont fournies par la ligne N, c'est ce qui est fait durant la comparaison des valeurs de la ligne N avec celles de la ligne j.

III.2.2 Complexité de l'algorithme de Guo & Hwang :

Passons, maintenant, à la vérification de la complexité temporelle de cette implémentation. Le tableau X peut être construit dans un temps *O(n)* puisqu'on a n éléments dans la séquence j. Pour construire le tableau dynamique Y, on doit passer par les *O(n)* éléments de I, puisque m qui est la longueur de I est de l'ordre *O(n)*.

Puisque les entrées dans chaque ligne j et ligne N sont ordonnées, en commençant par comparer les entrées maximales des deux lignes, chaque comparaison élimine une entrée d'autres comparaisons, car si on compare l'exécution de l'implémentation de Pevzner&Waterman avec celle de Hwang&Guo nous remarquons que cette dernière implémentation ne traite pas toutes les paires ce qui représente un gain considérable en temps d'exécution, tandis que la deuxième passe par toute les paires. En plus l'ordre des indices nous permet de sauter directement vers la prochaine position du même nucléotide sans vérifier si les positions qui lui succèdent lui sont égales.

Puisqu'il y a au plus 2*L* entrées dans les deux lignes, ça prend *O(L)* temps pour localiser les entrées $\{n_i\}$ de la ligne N. L'insertion de la colonne de n_i (et probablement la suppression d'une colonne) prend un temps constant, car c'est une opération qui ne dépend pas de la taille des séquences donc c'est le même coût pour toutes les colonnes.

La fonction de retour arrière (le backtracking) a besoin d'être mise à jour au plus L fois, et cela prend un temps constant pour la mettre à jour, cela veut dire qu'on fait appel à cette fonction au maximum L fois, et que la mise à jour de sa valeur prends un temps constant qui ne dépend que du nombre de chaînes estimé à L chaînes.

Ainsi le traitement de chaque symbole de I prend *O(L)* temps car pour chaque symbole Ni de I, on doit chercher parmi les n symboles de J dans les L chaînes, la position du symbole Nj qui lui correspond. La construction du tableau Y prend *O(nL)* temps due aux *O(n)* éléments à parcourir et chaque élément à traiter dans *O(L)* temps. Donc on a un algorithme ayant un temps d'ordre *O(nL).* Il faut noter que les tableaux X et Y peuvent être construits dans un espace d'ordre *O(n)* parce que le traitement des deux tableaux s'effectue sur les n éléments de la séquence J et *m=O(n)*.

III.3 Conclusion :

Le but du remplacement des paires est d'arriver à les ordonner de telle façon qu'elles soient $\prec -$ comparables, ce qui fait que les index i et j de ces paires sont dans un ordre croissant.

L'utilité de la fonction b est de s'assurer que la paire d'avant n'est pas $\prec^*$ avec la paire actuelle.

Nous avons vu que l'implémentation de H&G ne fait que rechercher le PLCS, sans tenir compte du reste des symboles, ni de l'alignement des symboles. Donc, cet algorithme n'est efficace que pour trouver la plus longue sous-séquence commune à deux séquences.

Chapitre IV

Implémentation et expérimentation

Chapitre IV : Implémentation et expérimentation

Après avoir étudié les différents algorithmes de recherche du PLSC, nous avons décidé d'implémenter l'algorithme de Hwang et Guo ainsi que l'algorithme de Smith et Waterman, pour pouvoir les tester sur des exemples de séquences de différentes longueurs et comparer leurs résultats.

IV.1 : Environnement de Développement :

Pour implémenter ces deux algorithmes, nous avons utilisé l'outil de développement Borland Delphi 5(Entreprise Edition) qui offre une bonne interface de développement ainsi qu'une riche bibliothèque de fonctions de calculs et de traitement des données. Ce travail est effectué sur un système d'exploitation Windows. Pour pouvoir utiliser cette application sur un système Linux, nous devons utiliser Kylix qui est la une version Delphi adaptée à Linux.

IV.2 : Interface de l'application :

L'interface de l'application (voir figure 5.1) est simple et nous permet de sélectionner deux fichiers texte contenant les séquences à comparer, ou de les générer directement en indiquant leur longueur.

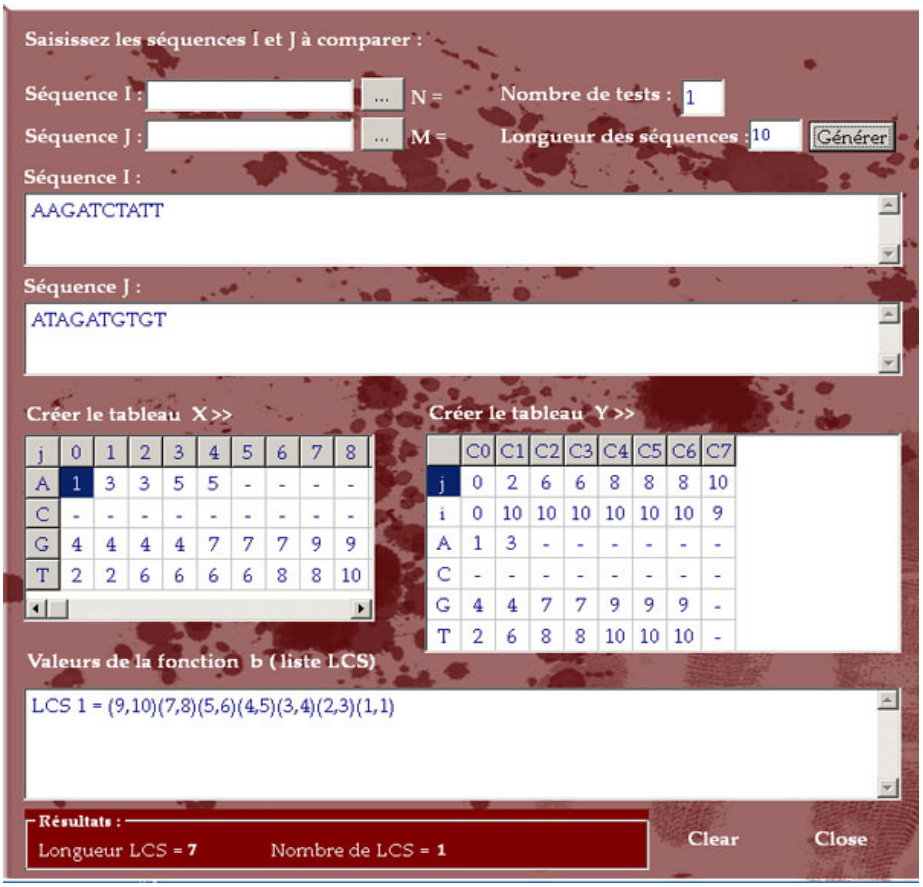


Figure 4.1 : Interface de l'application de recherche du PLCS

La comparaison des deux séquences est déclenché en deux parties, la première consiste à créer le tableau X et la seconde à créer le tableau Y, calculer les valeurs de la fonction b et ainsi constituer les paires des PLCS.

Le résultat est représenté par l'affichage du nombre des PLCS, leurs longueurs ainsi que les paires les constituant.

IV.3 : Exemples de Test :

Une fois l'application terminée, nous avons procédé à un ensemble de tests de comparaison de séquences de différentes longueurs (100, 1000, 10000, 100000 et 1000000). Par la suite nous avons apporté des modifications sur ces séquences à raison de 5%, 10%, 20% du nombre totales de leurs symboles pour les comparer avec les séquences d'origine.

Exemple : soit I1 une séquence de 100 symboles :

I1=ACCTGTCGGTGATGCTAACCGTCTGACTTACGTGGACTCCAAATGCTCCGAA
ATCGCTTAGTGCCTGAATACTGCCATGCACCTATGTTACGGCTACTCG

La séquence I2 obtenue par la modification de 5% de la séquence I1 est la suivante :

I2=ACCTGTCGGTGATGCTA**T**CCGTCTGACTTACGTGGACT**G**CAAATGCTCCGAA
ATCGCTT**GA**TGCCTGAATACTGCCATG**A**ACCTATGTTACGGCTACTCG

Donc le test sera appliqué sur ces deux séquences I1 et I2 à fin de trouver la plus longue sous séquence commune à I1 et I2.

Notons que ces tests sont effectués sur un PC ayant une vitesse de processeur de l'ordre de 2.4 Ghz et dotée d'une RAM de 1Go.

Après plusieurs tests sur des séquences différentes, nous avons pu noter les durées de chaque comparaison et la longueur du PLCS trouvée. Nous présentons ces valeurs dans les tableaux et graphes suivants :

Séquences	Longueurs des séquences	Nombre de tests appliqués	Type des Séquences comparées			
			Séquences Aléatoires et indépendantes	Séquence aléatoire + séquence modifiée à 5%	Séquence aléatoire + séquence modifiée à 10%	Séquence aléatoire + séquence modifiée à 20%
S1, S2	100	100	462	412	410	402
S2, S1				382	391	400
S3, S4			451	386	405	394
S4, S3				380	424	414
S5, S6			487	387	385	424
S6, S5				364	382	409
S7, S8	200	100	1667	1359	1425	1515
S8, S7				1341	1411	1508
S9, S10			1619	1359	1388	1437
S10, S9				1453	1388	1474
S11, S12			1606	1605	1448	1445
S12, S11				1368	1420	1446
S13, S14	500	100	9794	7873	8114	8493
S14, S13				7930	8272	8683
S15, S16			9575	7935	8067	8594
S16, S15				7870	8212	8708
S17, S18			9497	7776	7957	8059
S18, S17				7772	7968	8422
S20, S21	1000	50	36833	29240	30433	32308
S21, S20				29720	30408	32995
S22, S23			36484	32306	30575	32176
S23, S22				29818	30653	32168
S24, S25			36824	29761	30753	36824
S25, S24				29900	30659	32525
S26, S27	5000	10	835891	660868	680589	719210
S27, S26				663003	681844	721732
S28, S29			815746	663123	684577	717437
S29, S28				664849	685677	717438
S36, S37			815489	661622	682265	719639
S37, S36				664988	681079	716878
S30, S31	10000	10	3365349	2630712	2721835	2866050
S31, S30				2637795	2722733	2880007
S32, S33			3232697	2632757	2716923	2864358
S33, S32				2640251	2724639	2875960
S38, S39			3350248	2689293	2731826	2855644
S39, S38				2642962	2713434	2864089

Tableau 4.1 : Durées moyennes de recherche des PLCS par l'implémentation de Smith et Waterman.

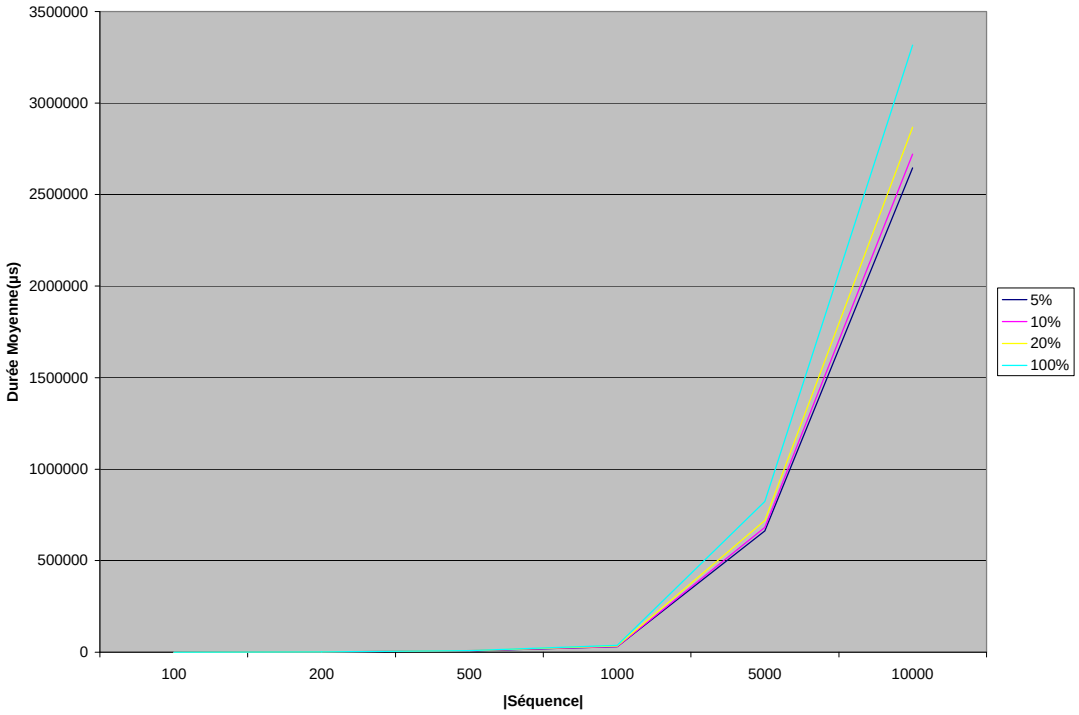


Figure 4.2 : Graphe de l'évolution des durées moyennes de recherche des PLCS par l'implémentation de Smith et Waterman.

Séquences	Longueurs des séquences	Nombre de tests appliqués	Type des Séquences comparées			
			Séquences Aléatoires et indépendantes	Séquence aléatoire + séquence modifiée à 5%	Séquence aléatoire + séquence modifiée à 10%	Séquence aléatoire + séquence modifiée à 20%
S1, S2	100	100	295	264	262	257
S2, S1				244	250	256
S3, S4			289	247	259	252
S4, S3				243	271	265
S5, S6			312	248	246	272
S6, S5				233	245	261
S7, S8	200	100	1067	870	912	970
S8, S7				858	903	965
S9, S10			1036	870	888	920
S10, S9				930	888	943
S11, S12			1028	1027	927	925
S12, S11				875	909	926
S13, S14	500	100	6268	5039	5193	5436
S14, S13				5075	5294	5557
S15, S16			6128	5078	5163	5500
S16, S15				5037	5256	5573
S17, S18			6078	4977	5092	5157

S18, S17				4974	5100	5390
S20, S21	1000	50	23573	18714	19477	20677
S21, S20				19021	19461	21117
S22, S23			23350	20676	19568	20593
S23, S22				19083	19618	20587
S24, S25			23567	19047	19682	23567
S25, S24				19136	19622	20816
S26, S27	5000	10	534970	422955	435577	460294
S27, S26				424322	436380	461908
S28, S29			522077	424399	438129	459160
S29, S28				425503	438833	459160
S36, S37			521913	423438	436650	460569
S37, S36				425592	435891	458802
S30, S31	10000	10	2153824	1683656	1741975	1834272
S31, S30				1688189	1742549	1843204
S32, S33			2068926	1684964	1738831	1833189
S33, S32				1689760	1743769	1840614
S38, S39			2144182	1721147	1748369	1827612
S39, S38				1691496	1736598	1833017

Tableau 4.2 : Durées moyennes de recherche des PLCS par l'implémentation de Hwang et Guo.

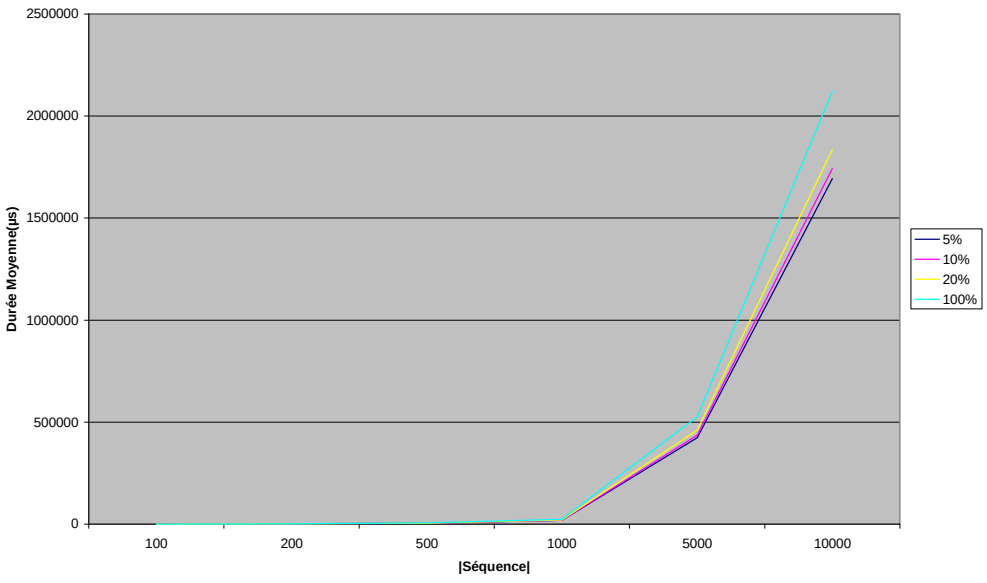


Figure 4.3 : Graphe de l'évolution des durées moyennes de recherche des PLCS par l'implémentation de Hwang et Guo

Séquences	Longueurs des séquences	Nombre de tests appliqués	Type des Séquences comparées			
			Séquences Aléatoires et indépendantes	Séquence aléatoire + séquence modifiée à 5%	Séquence aléatoire + séquence modifiée à 10%	Séquence aléatoire + séquence modifiée à 20%
S1, S2	100	100	108	234	127	135
S2, S1				106	101	82
S3, S4			94	96	91	77
S4, S3				81	124	243
S5, S6			172	82	78	108
S6, S5				81	65	85
S7, S8	200	100	268	277	242	353
S8, S7				246	272	290
S9, S10			244	228	279	250
S10, S9				397	301	278
S11, S12			291	286	411	233
S12, S11				328	273	271
S13, S14	500	100	793	820	797	863
S14, S13				723	913	913
S15, S16			747	813	772	820
S16, S15				961	864	883
S17, S18			736	932	799	741
S18, S17				1057	736	993
S20, S21	1000	50	2062	2569	1905	2749
S21, S20				2429	2280	3505
S22, S23			2460	5585	2462	2228
S23, S22				1888	1684	2059
S24, S25			2477	2710	2117	2252
S25, S24				1817	1963	1841
S26, S27	5000	10	56672	3655	4783	5852
S27, S26				6418	5034	5554
S28, S29			5858	5477	4667	5088
S29, S28				4449	5419	4409
S36, S37			14453	5387	2331	1831
S37, S36				5233	3004	2828
S30, S31	10000	10	415410	22681	10077	20767
S31, S30				12295	39242	25738
S32, S33			14033	24097	7264	4561
S33, S32				12251	11668	8272
S38, S39			361280	176942	4272	4908
S39, S38				6326	3704	15528

Tableau 4.3 : Ecarts type des durées de recherche des PLCS par l'implémentation de Smith et Waterman.

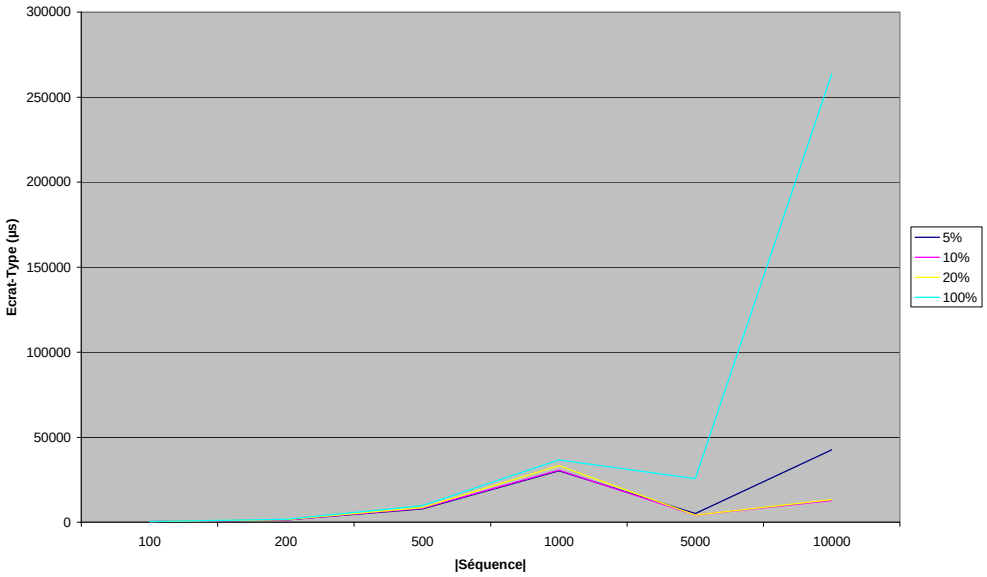


Figure 4.4 : Graphe de l'évolution des ecarts type des durées de recherche des PLCS par l'implémentation de Smith et Waterman.

Séquences	Longueurs des séquences	Nombre de tests appliqués	Type des Séquences comparées			
			Séquences Aléatoires et indépendantes	Séquence aléatoire + séquence modifiée à 5%	Séquence aléatoire + séquence modifiée à 10%	Séquence aléatoire + séquence modifiée à 20%
S1, S2	100	100	69	150	81	86
S2, S1				68	65	52
S3, S4			60	61	58	49
S4, S3				52	79	155
S5, S6			110	53	50	69
S6, S5				52	41	55
S7, S8	200	100	172	177	155	226
S8, S7				158	174	186
S9, S10			156	146	179	160
S10, S9				254	193	178
S11, S12			186	183	263	149
S12, S11				210	175	174
S13, S14	500	100	508	525	510	552
S14, S13				463	584	584
S15, S16			478	520	494	525
S16, S15				615	553	565
S17, S18			471	596	511	474
S18, S17				676	471	636
S20, S21	1000	50	1320	1644	1219	1759

S21, S20				1555	1459	2243
S22, S23			1574	3575	1576	1426
S23, S22				1208	1078	1318
S24, S25			1585	1735	1355	1441
S25, S24				1163	1257	1178
S26, S27	5000	10	36270	2339	3061	3745
S27, S26				4107	3221	3554
S28, S29			3749	3506	2987	3256
S29, S28				2847	3468	2822
S36, S37			9250	3448	1492	1172
S37, S36				3349	1923	1810
S30, S31	10000	10	265822	14516	6450	13291
S31, S30				7869	25115	16472
S32, S33			8981	15422	4642	2919
S33, S32				7841	7468	5294
S38, S39			231219	111976	2734	3141
S39, S38				4048	2371	9938

Tableau 4.4 : Ecarts type des durées de recherche des PLCS par l'implémentation de Hwang et Guo.

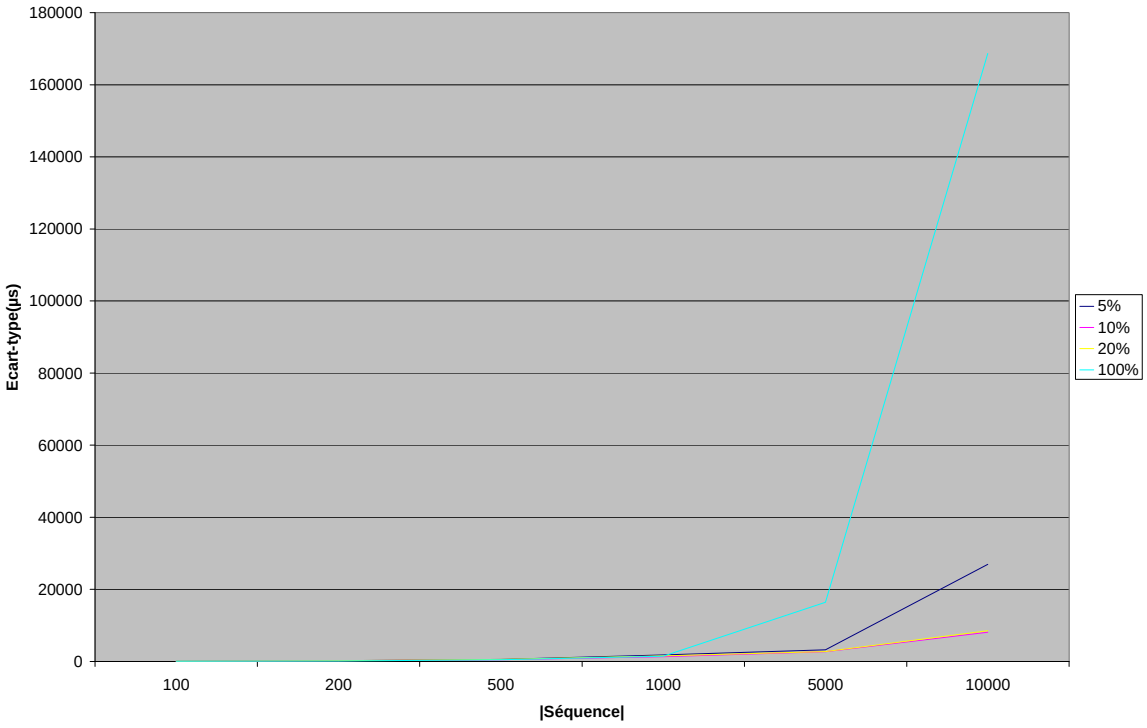


Figure 4.5 : Graphe de l'évolution des écarts type des durées de recherche des PLCS par l'implémentation de Hwang et Guo.

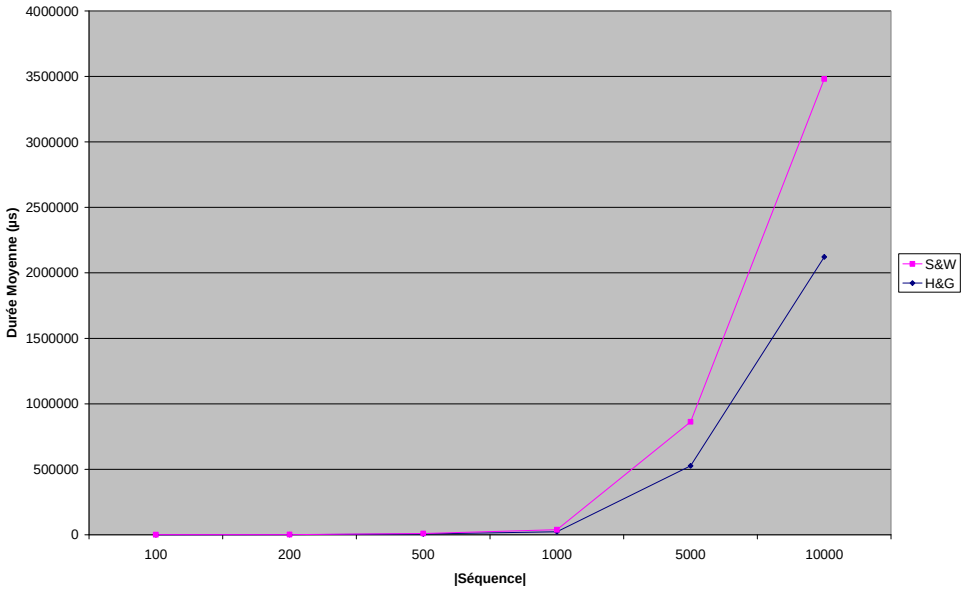


Figure 4.6 : Graphe de comparaison des durées de recherches du PLSC entre les deux implémentations.

Séquences	Longueurs des séquences	Type des Séquences comparées			
		Séquences Aléatoires et indépendantes	Séquence aléatoire + séquence modifiée à 5%	Séquence aléatoire + séquence modifiée à 10%	Séquence aléatoire + séquence modifiée à 20%
S1, S2	100	64	96	93	88
S2, S1			96	94	87
S3, S4		63	97	94	88
S4, S3			97	95	89
S5, S6		63	97	94	89
S6, S5			97	93	88
S7, S8	200	129	192	184	171
S8, S7			192	187	172
S9, S10		129	193	188	178
S10, S9			192	186	176
S11, S12		123	195	187	177
S12, S11			194	184	175
S13, S14	500	323	481	466	446
S14, S13			482	469	439
S15, S16		319	481	467	436
S16, S15			484	468	438
S17, S18		315	484	467	438
S18, S17			479	465	441
S20, S21	1000	643	972	936	874
S21, S20			959	938	887
S22, S23		650	970	932	875
S23, S22			966	935	880
S24, S25		645	966	934	881
S25, S24			961	938	868

S26, S27	5000	3258	4815	4644	4362
S27, S26			4821	4660	4384
S28, S29		3261	4818	4663	4373
S29, S28			4804	4661	4386
S36, S37		3259	4822	4655	4387
S37, S36			4812	4666	4371
S30, S31	10000	6506	9654	9310	8782
S31, S30			9654	9329	8776
S32, S33		6524	9642	9309	8755
S33, S32			9651	9302	8744
S38, S39		6523	9664	9308	8770
S39, S38			9638	9326	8748

Tableau 4.5 : Longueur des PLSC trouvées par les deux implémentations.

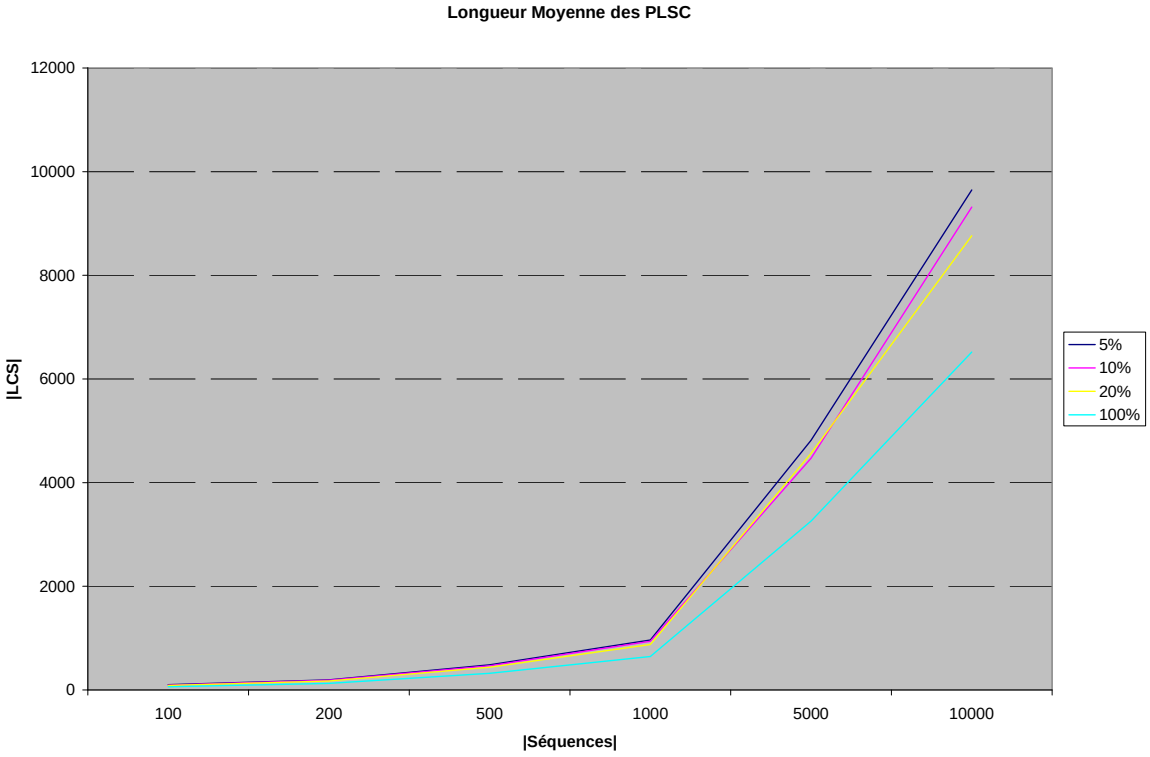


Figure 4.7 : Graphe de l'évolution de la longueur du PLCS par rapport à la longueur des séquences comparées

IV.3 : Conclusion :

Dans ce chapitre nous avons exposé les deux applications issues des implémentations des algorithmes de Hwang et Guo ainsi que de Smith et Waterman, que nous avons tester sur plusieurs séquences à fin de comprendre le rapport entre le temps d'exécution et la longueur des séquences traitées ainsi que la longueur des sous-séquences communes obtenues. La comparaison de ces résultats nous a montré que l'implémentation de Huang et Guo est plus rapide que celle de Smith et Waterman.

Chapitre V

Extension de l'implémentation

Chapitre V : Extension de l'implémentation

V.1 : Introduction générale :

Après avoir étudié quelques algorithmes de recherche de la plus longue sous-séquences commune à deux séquences, dans les deux approches de programmation (dynamique et non dynamique), nous avons effectué des tentatives pour améliorer l'implémentation de Hwang et Guo, et proposé des solutions pour d'autres problèmes.

La première tentative pour résoudre le problème d'alignement des séquences obtenues par Hwang et Guo (H&G), consistait à étudier d'abord l'algorithme de Hirschberg lequel, après avoir calculé le score d'alignement, cherche l'alignement de ces deux séquences.

Ce travail ne peut pas se faire simultanément avec celui de Hwang et Guo car nous avons tenté de fusionner leurs deux algorithmes.

Or, nous savons que l'implémentation de H&G ne fait que chercher la PLCS, sans tenir compte du reste des symboles. Tandis que l'algorithme de Hirschberg permet d'évaluer les coûts d'alignement de n'importe quel symbole avec la séquence J avec la possibilité de choisir l'alignement qui nous convient. Donc, l'algorithme de H&G n'est efficace que pour trouver la plus longue sous-séquence commune à deux séquences.

La deuxième tentative, présentée ci-dessous, concerne trois solutions éventuelles pour la recherche de la plus longue sous-séquence commune à deux séquences de mots, et d'aligner des séquences nucléiques après que l'on ait trouvé et sélectionné une PLCS, par l'implémentation de H&G ainsi que son extension pour aligner des mots.

V.2 : Extension de l'implémentation de H&G pour traiter des mots (Application sur des séquences nucléiques) :

V.2.1 Partie 1 :

Définitions de base

Pour pouvoir trouver des mots communs à deux séquences, on a besoin :

1/ de la relation $\prec$
2/ de connaître la valeur de Δ_i qui est définit comme suit :

Soient $p_1= (i_1,j_1)$ et $p_2= (i_2,j_2)$ deux éléments aléatoires dans P:

$\Delta_i = \Delta_i (p_1,p_2) = i_2-i_1$

$\Delta j = \Delta j (p_1, p_2) = j_2-j_1$

$\Delta = \Delta (p_1, p_2) = (\Delta_i , \Delta j)$

Alors nous dirons que $p_1 \prec p_2$ *si* $\Delta_i>0 \wedge \Delta j>0$

Définition des objectifs

Après avoir présenté les définitions de bases que nous utiliserons par la suite, nous devons définir un but à atteindre, et c'est selon ce but que nous modifierons les définitions citées ci-dessus.

Nous avons recensé deux buts différents, dont la différence se situe au niveau de la forme du mot qu'on doit chercher dans la deuxième séquence, à savoir :

Cas A : si on veut des alignements de type :

Toutes deux paires successives doivent vérifier la condition : $\Delta_i=1$ et $\Delta j=1$

Cas B : si on veut des alignements de type :

Toutes deux paires successives doivent vérifier la condition : $\Delta_i=1$ et $\Delta j>0$

Exemple à appliquer : I = ACTGGC et J=TACTGGCCA

	1	2	3	4	5	6
I=	A	C	T	G	G	C

	1	2	3	4	5	6	7	8	9
J=	T	A	C	T	G	G	C	C	A

Construisons l'ensemble P qui contient toutes les paires de match possibles.

	P1	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*	*P13*	*P14*
P={(1,2), (1,9), (2,3), (2,7), (2,8), (3,1), (3,4), (4,5), (4,6), (5,5), (5,6), (6,3), (6,7), (6,8)}														

Si on cherche le cas B c'est-à-dire : $\Delta i = 1, \Delta j = 1$

$p_1 \prec p_3$, $p_3 \prec p_7$, $p_7 \prec p_8$, $p_8 \prec p_{11}$, $p_{11} \prec p_{13}$

{(1,2), (2,3), (3,4), (4,5), (5,6), (6,7)}

Si on cherche le cas A c'est-à-dire : $\Delta i = 1, \Delta j > 0$

$p_1 \prec p_3$, $p_1 \prec p_4$, $p_1 \prec p_5$, $p_3 \prec p_7$, $p_6 \prec p_8$, $p_6 \prec p_3$, $p_7 \prec p_8$, $p_7 \prec p_9$, $p_8 \prec p_{11}$, $p_{10} \prec p_{13}$, $p_{10} \prec p_{14}$, $p_{11} \prec p_{13}$, $p_{11} \prec p_{14}$

on remarque que le cas B est inclut dans le cas A, et que le cas A nous donnes des alignement globaux alors que le cas B nous donne des alignements locaux.

Pour le cas A :

L'exemple pris est appliqué sur A donc on obtient des paires dont :

$\forall p1, p2 \in P$, $p_1 \prec p_2 \Leftrightarrow \Delta_i = 1$ et $\Delta j = 1$

Pour le cas B : on a le résultat directement mais doit-on trouver une autre explication que le fait que

$b(p_2) = p_1$ ssi $\Delta(p1, p2) = (1,1)$

Et

$\forall p1, p2 \in C$:

Soit $p_1 \prec p_2$ soit $p_2 \prec p_1 \Rightarrow \Delta i = 1, \Delta j \geq 1$

et que ça suffit, sinon on calcule les chaînes, ce qui constitue une perte de temps.

V.2.2 : Partie 2 :

Problématique

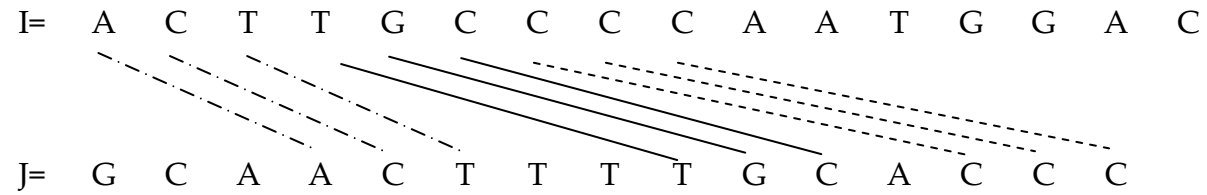
L'algorithme de Hwang et Guo permet de trouver la plus longue sous-séquence commune (PLCS) à deux séquences de symboles appartenant à un alphabet. Cette PLCS est représentée par un ensemble de paires de positions.

Nous voulons modifier l'implémentation de Hwang et Guo pour pouvoir trouver la plus longue sous-séquence commune à deux séquences de mots et non de symboles, car dans les cas pratiques l'alphabet peut-être des mots ou des codons

Fonctionnement

Avant d'entamer ce travail, il est nécessaire, en premier lieu, de se mettre d'accord sur le fait que l'on veut trouver des mots complets (codons) et non des bribes de mots. C'est-à-dire si nous cherchons le codon TCC, nous devons trouver TCC et non TC ou CC.

Prenons l'exemple suivant où nous avons deux séquences I=ACTTGCCCCAATGGAC et J= GCAACTTTTGCACCC.



On suppose que la première séquence I est originale, c'est-à-dire ne contient pas de mutations et qu'on veut chercher ses mots dans la seconde séquence J, pour pouvoir détecter les mutations qu'a subit cette séquence.

Question 1 :

Doit-on constituer un mot à partir d'un dictionnaire, c'est-à-dire dans la liste des codons ?

Car il se peut que le premier symbole de la séquence ne soit pas le début du premier mot, c'est le cas où la séquence a été choisie au hasard et, donc, le premier triplet de la séquence I n'aura pas d'équivalent dans la séquence J.

Donc ce dictionnaire nous permet de trouver le bon repère sur une séquence pour commencer le traitement par un mot correct.

Idée 1 :

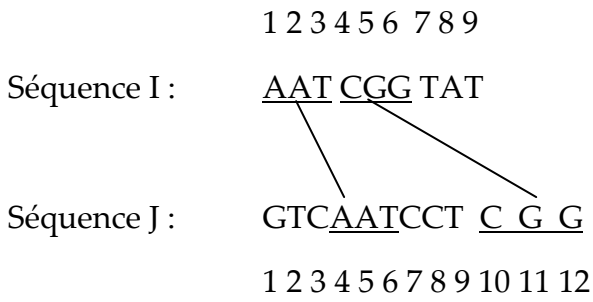
La recherche du PLCS pourra se faire sans prendre en compte la longueur des mots appariés et, à la fin de la recherche, seront éliminés les mots ayant une longueur inférieure à trois (03) symboles, ce qui serait une perte de temps au regard de ce double travail.

Donc, il serait bon de concentrer la recherche sur des mots de trois symboles dès le début.

Un autre critère pour vérifier qu'une séquence est correcte, est d'avoir sa longueur modulo 3 (taille d'un codon) égale à 0, ce qui nous permet de supposer que l'extraction de la séquence est correcte et que ses codons sont complets.

Aussi, il reste à trouver les mots de la séquence I dans la séquence J pour déterminer les paires d'index (i,j) où I[i]=J[j]. Un index correspond à la position du premier symbole du mot trouvé.

Exemple : Dans cette exemple les paires d'index sont (1,4) et (4, 10).

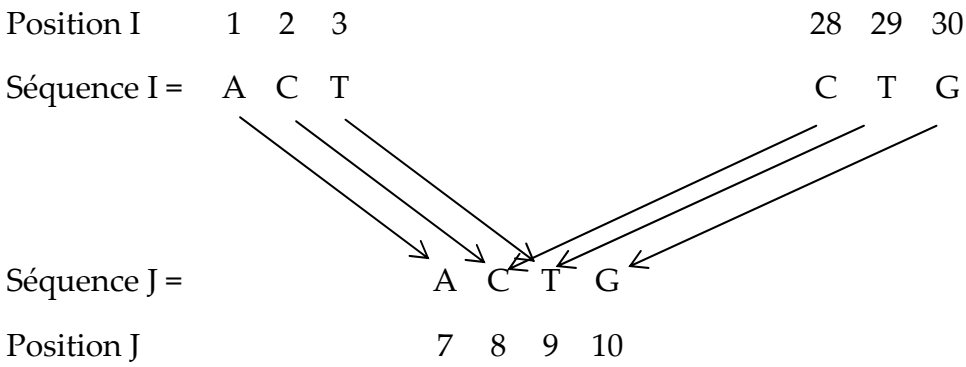


Idée 2 :

Si I(i)=J(j), I(i+1)=J(j+1) et I(i+2)=J(j+2), dans ce cas, ce mot sera retenu, sinon il faut passer au mot suivant, c'est-à-dire à partir de la position j+3.

Donc si pour un mot de la séquence I on lui trouve un ou plusieurs mots identiques dans la séquence J, seul l'indice I sera utilisé, en l'occurrence, on retiendra la paire (i,j) pour ne pas tomber dans le piège, quoique scientifiquement, cela est exact.

Exemple :



On doit retenir les paires (1,7) et (28,8) ; aussi, il reste seulement à vérifier la condition entre les paires à traiter : $\Delta j \geq 3$

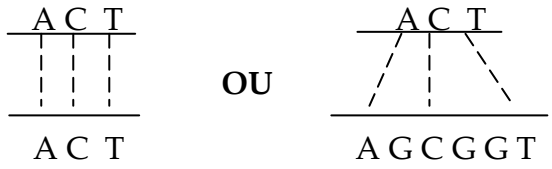
C'est-à-dire, si l'on a p1= (1,1) et p2= (1,4), nous ne retiendrons p2 que parce que $\Delta j \geq 3$ (ici 4 -1 =3).

De ce fait, la première étape consiste à chercher les paires de correspondances entre les symboles des deux séquences I et J. Si l'on ne trouve pas de I(i)=J(j), alors il faut passer à i+3.

Mais l'implémentation dépend des résultats voulus et, dans ce cas, une seconde question se pose :

Sommes-nous à rechercher dans la séquence J des codons ou des symboles constituant le codon ?

Pour mieux comprendre cette question, il est bon de l'illustrer par la figure suivante :



Cas A **Cas B**

Pour le cas A, il faut que $\Delta i = 1$ et $\Delta j = 1$.
Pour le cas B, il faut que $\Delta i = 1$ et $\Delta j \geq 1$.

V.3 Comment modifier l'algorithme de Hwang et Guo pour traiter des mots :

Le travail que nous allons présenter s'articule autour des possibles modifications qu'on doit apporter à l'implémentation de Hwang et Guo pour la recherche de la plus longue sous-séquence commune à deux séquences de mots.

Durant ce travail nous proposerons des solutions qui se basent sur des fonctions système, c'est à dire propres au langage de programmation utilisé, et que l'on pourra modifier par la suite pour devenir plus formel. Donc tout ce que nous allons présenter comme structure de données reflète l'implémentation qui nous a permit d'atteindre la bonne solution.

Pour commencer nous devons se lancer sur la base d'un ensemble de suppositions telles :

Le traitement se fera sur des mots et non sur des caractères (lettres de l'alphabet).

L'ordre lexicographique des mots est établit par des fonctions système « LIKE » et « = » qui nous permettent de savoir si un mot est égal à un autre mot ou est son facteur (gauche ou droit). Cet ordre est nécessaire pour arriver à appliquer les relations d'ordre $\prec$ et $\prec^*$.

Soient I=i1, i2, i3,….in et J=j1, j2, j3…jm deux séquences de mots, et deux mots $w \in I$ et $w' \in J$, nous définissons les différentes opérations d'édition réalisables sur I et J, à savoir :

Insertion d'un gap dans *I:* $I= i_1\ i_2\ -\ i_3 \ldots i_n$

$$J= j_1\ j_2\ w'\ j_3 \ldots j_m$$

Insertion d'un gap dans *J*: $I= i_1\ i_2\ w\ i_3 \ldots i_n$

$$J= j_1\ j_2\ -\ j_3 \ldots j_m$$

Identité (match) $w=w'$: $I= i_1\ i_2\ w\ i_3 \ldots i_n$

$$J= j_1\ j_2\ w'\ j_3 \ldots j_m$$

Similarité : $I=i_1\ i_2\ w\ i_3 \ldots i_n$

$J=j_1\ j_2\ w'\ j_3 \ldots j_m$ tel que $w \subseteq w'$ **ou** $w' \subseteq w$, c'est à dire l'un est un facteur de l'autre.

Une question se pose : **Est-ce que ce dernier cas est pris en compte puisqu'on a dit qu'un mot est pris comme une seule entité ?**

En supposant que la similarité est prise en considération, et d'après la relation $\prec^*$, on doit avoir un probable croisement. Comment est-ce possible ?

Cette dernière supposition est réalisable, car on ne va pas prendre des caractères, mais des mots entiers et c'est leurs positions qui joueront un rôle et non leur contenu donc on oubli les « LIKE ». Ainsi on garde les relations $\prec$ et $\prec^*$ telle qu'elles sont.

Prenons l'exemple suivant pour mieux comprendre le but qu'on doit atteindre.

Exemple : soient les deux séquences I = « la mer est bleue » et J = « le bleu est la couleur de mer »

Donc notre PLCS sera composé des deux mots la et mer et l'alignement de I avec J sera de la forme :

I : – – – la – – mer est bleue

J : le bleu est la couleur de mer – –

Alors pour notre implémentation, nous avons utilisé des tableaux de type « string».

Dans l'étape qui suit, nous supposons qu'on veut aller dans plus de détail en impliquant les mots semblables. Dans notre exemple cité ci-dessus, on prendra en compte les mots « bleu » de I et « bleue » de J, car le premier mot est facteur gauche du second mot. Cette comparaison est réalisable grâce à la commande « LIKE », donc les relations ne changent toujours pas.

Le problème qui se pose maintenant, est que nous n'avons pas un alphabet fini, car nous manipulons des mots, donc soit on détaille la description des mots en allant jusqu'à comparer les mots, caractère par caractère, et donc on aura un tableau X ayant plus de 26 lignes (26 lettres de l'alphabet) ce qui consommera beaucoup de temps et d'espace mémoire. Sinon en lançant le traitement des deux séquences, on recense les mots des deux séquences.

On suppose que deux mots successifs sont séparés par un espace blanc ou un point.

Maintenant que l'alphabet est construit, nous pouvons procéder à la construction du tableau X. Soit E l'alphabet à utiliser, alors E= {la, mer, est, bleue, le, bleu, couleur, de}.

Il serait aberrant de garder tous les mots de la séquence J, car le PLCS ne contiendra que les mots w tels que $w \in I$ et $w \in J$ .Ainsi l'ensemble E se réduira à l'ensemble {la, mer, est, bleu}

1ère étape : Construction du tableau X :

La construction du tableau X nécessite :
Un nombre de colonnes = (nombre de mots de J) +1 ;
Un nombre de lignes = (nombre de mots de $I \cap J$) ; sans doublons.

Pour arriver à ce résultat il faut :
1/ Obtenir les mots des deux séquences :

Tabi (indi) : les mots de la séquence I avec doublons

0	1	2	3
la	mer	est	bleue

Tabj (indj) : les mots de la séquence J avec doublons

0	1	2	3	4	5	6
le	bleu	est	la	couleur	de	mer

Alpha(ind) : les mots de $I \cup J$ =A*

0	1	2	3	4	5	6	7
la	mer	est	bleue	le	bleu	couleur	de

2/ Il faut garder dans J les mots appartenants à I : tableau Reste (indest).

0	1	2	3
bleu	est	la	mer

3/ Retirer les doublons de Reste : tableau Change. Dans notre exemple le tableau Change a le même contenu que Reste.

4/ Il faut ensuite trouver les positions (indices) des mots de I dans tabj.

5/ L'alphabet de J est trouvée donc on peut construire le tableau X.

6/ Dans tabj, chaque case (indi+1) est la position du mot dans J puisque l'indice d'un tableau débute de zéro. Il faut prendre les mots de Change et les chercher dans tabj, dans le sens inverse (down-to), et si le mot est trouvé, on met les indices dans le tableau X.

Donc le tableau X aura le contenu final suivant :

J	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
La	-	-	-	-	4	4	4	4
Mer	-	7	7	7	7	7	7	7
Est	-	-	-	-	-	3	3	3
bleu	-	-	-	-	-	-	2	2

2ème étape : Construction du tableau Y :

Le nombre de ligne est égale au (nombre des mots de $I \cap J$)+ 3. (3 positions pour C_k, j, i).

Pour connaître l'indice du mot en cour de traitement : on ne fait le traitement que sur les mots de $I \cap J$, donc :

Si un mot $w \in I$ et $w \notin J$ alors Indice (w)=0 (par rapport au tableau Change).

Si un mot $w \in I$ et $w \in J$ alors Indice (w)=k tel que Change[k]=w.

Le traitement des paires se fait ensuite comme dans l'implémentation de Hwang et Guo. Nous obtenons à la fin du traitement un tableau contenant les paires formant le PLCS.

V.4 Problème d'alignement pour les implémentations de Hwang&Guo et Pevzner et Waterman :

Après leur avoir appliqué l'implémentation de P&W ou celle de H&G et obtenu la plus longue sous-séquence (PLCS) commune à ces deux séquences, nous savons que la solution d'alignement de deux séquences est la même ; et que la seule différence entre ces deux implémentations est la rapidité de l'implémentation de H&W.

Les résultats obtenus le sont sous forme d'un ensemble de paires classées par chaîne, chaque paire appartenant à une seule chaîne. Donc, si l'on sélectionne les paires formant une PLCS, on ne prendra en compte que les index de ces paires et on ignorera les paires non retenues.

Ainsi, pour aligner les paires de ces deux séquences, on doit :

1- Fixer les index des paires formant la PLCS.

2- Compléter ou aligner le reste des paires selon la logique qui dit :

2.1 - Un nucléotide A fait toujours face à un nucléotide T.

2.2 - Un nucléotide C fait toujours face à un nucléotide G.

De ce fait, la question suivante est posée:

Sachant que des mutations peuvent survenir sur une séquence d'ADN, est-il possible d'avoir un cas où, par exemple, un nucléotide A fait face à un nucléotide G ou C ?

La réponse est que cela n'est pas possible car la composition chimique de l'Adénine (nucléotide A) ne permet pas d'avoir un lien avec le Cytosine(C) ou le Guanine(G).

Mais dans le cas d'une comparaison de deux séquences de deux générations différentes, deux cas de figures se présentent :

les mutations de G ←→ T, G ←→ A, C ←→ T et C ←→ A sont possibles ; donc, durant l'alignement, la priorité est donnée aux mutations au lieu des gaps.

Ces mutations ne sont pas possibles, ou ne sont pas permises biologiquement (n'ont pas de sens biologique) et, alors dans ce cas on n'acceptera que les positions logiques A←→T et C ←→ G.

En se basant sur le deuxième cas et en prenant l'exemple suivant :

Soit deux séquences I= ACCAAT et J= AGGGGTT et en supposant que le PLCS se compose des deux paires ayant comme index (1,1) et (6,7).

L'alignement de ces deux séquences doit être de la forme suivante :

Séquence I	A	C	C	-	-	A	A	T
Séquence J	A	G	G	G	G	T	-	T

Figure 5.1 : Exemple d'alignement des séquences I et J.

Donc, l'on a trois types d'opérations :
Identité : (A, A), (T, T)
Indel : (-, G), (-, G), (A,-)
Mutation : (C, G), (C, G), (A, T)

En ayant la liste des paires du PLCS : (1,1), (6,7), l'algorithme suivant peut être utilisé pour aligner les symboles des deux séquences entre ces deux paires.

Algorithme de base :

Le traitement se fera en tenant compte des valeurs des paires formant le PLCS, qu'on met dans un tableau PLCS[L], tels que L est la longueur du PLCS :

Début
K :=1 ;
Tant que *PLCS[K]<> Nil* ***Alors***
Début
Position := PLCS[k] ; //paire PLCS actuelle : la paire (1,1)
    Tant que *pas arrivé à la prochaine paire du PLCS* ***faire***
        Aligner les symboles entre ces deux paires ;
    Fait ;
K :=K+1; // paire suivante du PLCS : (6,7)
Fin
Aligner le reste des symboles des deux séquences ;
Fin.

Remarque :

Lorsqu'on sélectionne des paires pour constituer notre PLCS, on ne prendra plus en considération les autres paires appartenant aux différentes chaînes, donc leurs valeurs d'identité ne sont plus prises en considération. C'est-à-dire, si l'on prend deux paires successives du PLCS, on alignera les symboles entre ces deux paires, sans considérer le fait qu'il y ait des positions de ces symboles qui forment des paires constituant l'une des chaînes calculées.

La proposition suivante est liée à la qualité biologique de notre alignement, car nous savons qu'il y a des alignements où on assigne un degré de pénalité pour tout gap inséré dans une séquence, pour limiter l'utilisation des gaps et pouvoir évaluer la qualité biologique des séquences à aligner, comme il existe des alignements où on ne tolère pas la présence de plusieurs gaps successifs, de sorte à obtenir un gain d'espace mémoire.

1 / Si on opte pour un minimum de gaps, on doit calculer la longueur des tranches. Une tranche est la séquence de symboles entre deux positions formant deux

paires successives du PLCS. Dans notre exemple, on a les index 1 et 6 de la séquence I qui appartiennent au PLCS, donc la tranche de I est formée des symboles appartenant à l'intervalle]1,6[, ici c'est [CCAA] .

Alors nous calculons les tranches de I et J entre chaque deux paires successives du PLCS, pour ensuite comparer leurs longueurs, à fin de savoir où l'on doit mettre le maximum de gaps.

Pour connaître la plus longue tranche, on calcule la tranche (I) (PLCSi+1 – PLCSi) et la tranche (J)=(PLCSj+1-PLCSj). Dans notre exemple : tranche (I)=6-1=5, tranche (J)=7-1=6

Donc si tranche (I) < tranche (J) alors le maximum de gaps se trouvera dans la tranche de I.

L'alignement des symboles des deux tranches respectera le même algorithme de base, c'est-à-dire en comparant deux paires successives (i1,j1) avec (i2,j2).

2/ Sinon, si nous optons pour une meilleure qualité biologique, il faut mettre en priorité la séquence la plus correcte puisque la séquence à comparer est soit la plus correcte, soit celle ayant subi des mutations ou erreurs. Donc en priorité on analysera les symboles de la bonne séquence ; ensuite, ceux restants de la séquence de mauvaise qualité.

Exemple :

Soient les deux séquences suivantes :

Bonne séquence : ACCTA

Mauvaise séquence : AGGGA

Alors nous prendrons en compte l'alignement suivant :

ACCT - A

AGG -GA

Au lieu de l'alignement suivant :

ACC-TA

AGGG-A

Maintenant, il faut se demander comment faire pour obtenir un alignement avec un sens biologiquement correct ?

L'algorithme d'alignement proposé se base sur 4 index, deux de la première paire (i1,j1) et deux de la seconde paire (i2,j2).Pour cela, on doit donner la priorité d'alignement à la première position i1 et j1, et si on arrive à un i1= PLCSi+1 ou j1= PLCSj+1 alors il prendra la valeur d'un gap « − » .

Pour mieux comprendre cette idée, nous prendrons un exemple sur lequel nous allons développer le corps de notre algorithme :

	PLCSi	i_1	i_2	i_3	$PLCS_{i+1}$
Séquence I :	**A**	**C**	**C**	**C**	**A**

Séquence J :	**A**	**G**	**T**	**A**
	PLCSj	j_1	j_2	$PLCS_{j+1}$

Nous appellerons un bloc, une suite de deux paires qui se succèdent.

Alors l'algorithme à appliquer sur ces deux séquences est le suivant :

Algorithme

Début

Aligner PLCSi avec PLCS j ;
Tant que *($i_2 \neq PLCS_{i+1}$) et ($j_2 \neq PLCS_{j+1}$)* ***faire***
Si $i_1 = j_2$ *alors*
Aligner (-,j_1) ;
Aligner (i_1,j_2) ;
Sinon si $j_1 = i_2$ *alors*
Aligner (i_1,-) ;
Aligner (i_2,j_1) ;
Sinon si $(i_1 \neq j_2)$ *et* $(j_1 \neq i_2)$ *alors*
Aligner (i_1,-);
Aligner (-,j_1) ;
Passer au bloc suivant ;
Si $(i_2 = PLCSi_{+1})$ *et* $(j_2 \neq PLCS_{j+1})$ *alors* i_2 :='-' ;
Si $(j_2 = PLCS_{j+1})$ *et* $(i_2 \neq PLCS_{i+1})$ *alors* j_2 :='-' ;
Fait ;

Fin ;

Cette solution n'est peut être utile que dans le cas où l'on veut étudier la possibilité de lier deux brins d'ADN, sous forme de la double hélice.

Dans le cas général, et pour pouvoir aligner deux symboles de deux séquences différentes, on devra utiliser une matrice des coûts d'alignement des 4 nucléotides, (méthode déjà vue dans la programmation dynamique).

Δ	A	C	G	T	-
A					
C					
G					
T					
-					

Chaque case de cette matrice représente le coût d'alignement de deux nucléotides, noté Δ[i,j].

On attribuera des coûts selon les spécificités biologiques du résultat qu'on veut obtenir ; par exemple nous attribuerons un coût élevé à un alignement de C avec T et un coût faible pour l'alignement de A avec T.

Nous prendrons les tranches entre les deux paires formant le PLCS et nous ferons appel à une fonction d'alignement dont, pour chaque couple de symboles (fa, gb) on a le choix entre les trois possibilités suivantes :

Aligner a avec b et f avec g

2- Aligner a avec − et f avec gb

3- Aligner b avec − et fa avec g

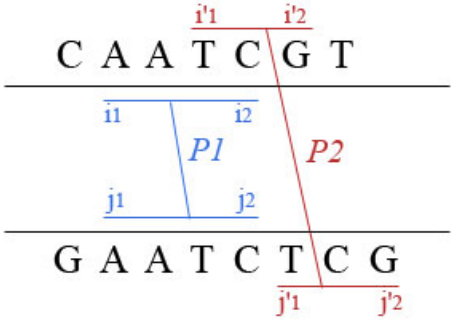
Il faut choisir l'un de ces trois alignements dont le coût est minimal c'est-à-dire utiliser une fonction δ pour l'évaluation de ces trois possibilités :

$$\delta(fa, gb) = \min \begin{cases} \delta(f, g) + \Delta[a, b] \\ \delta(fa, g) + \Delta[-, b] \\ \delta(f, gb) + \Delta[a, -] \end{cases}$$

V.5. Perspectives :

A partir de l'étude que nous venons de faire, plusieurs perspectives se présentent devant nous, dont celle d'étendre l'algorithme de Hwang et Guo à des textes non mots c'est à dire des cas où on cherche des séquences précises. Dans cette voie plusieurs cas se présentent et qui nous demande de reformuler les relations d'ordre définies par Hwang et Guo.

Le premier cas à étudier est celui où les mots recherchés se chevauchent, c'est-à-dire le cas où la fin du premier mot constitue le début du second mot.



Pour pouvoir trouver la PLCS, il faut reformuler la relation $\prec$ car nous ne pourrons pas avoir les deux paires p_1 et p_2 dans la même PLCS :

$p_1 \prec p_2$ Si $i_2 < i'_1$ et $j_2 < j'_1$ avec $p_1 = (i_1, i_2, j_1, j_2)$ et $p_2 = (i'_1, i'_2, j'_1, j'_2)$

V.6 Conclusion :

Durant cette étude, nous avons su que pour résoudre le problème de la plus longue sous-séquence commune à deux séquences, on doit choisir le but à atteindre, surtout lorsque l'on applique notre recherche sur des données biologiques, pour

pouvoir modifier les règles de bases de l'implémentation de Hwang et Guo, à fin de modifier le traitement et le converger vers le but voulu.

Les quelques propositions qui ont été faites se veulent un début pour des études plus approfondies et réalisables, du point de vue implémentation, et applicable sur des données, que ce soit des symboles ou des mots, appartenant à un alphabet biologique ou général.

Conclusion générale

Conclusion générale :

L'étude que nous avons menée concerne les principaux algorithmes de comparaison de deux séquences pour arriver à trouver la plus longue sous-séquence commune à ces deux séquences. Ces algorithmes font partie de l'approche de programmation dynamique ou de l'approche non-dynamique.

Pour le problème de la plus longue sous-séquence commune, l'approche de programmation dynamique exige un temps quadratique et un espace linéaire, alors que l'approche de programmation non-dynamique nécessite un temps d'ordre $O(n \log n)$ ou $O(Ln)$, qui est presque linéaire quand la longueur d'un PLCS est petite comparée à n comme c'est le cas de l'implémentation de l'algorithme du Primal-Dual proposée par Hwang et Guo qui s'effectue sur un alphabet faible réduit à quatre symboles {A,C,G,T}.

Cette implémentation permet de réduire le temps d'exécution en ne traitant qu'un nombre limité de paires de symboles. Ceci est possible par l'enregistrement au préalable des positions de chaque symbole de la deuxième séquence dans un tableau. Ce gain de temps est mis en évidence en exécutant l'implémentation de Pevzner et Weterman et l'implémentation de Hwang et Guo sur les mêmes séquences. La première exécution traite les 12 paires de l'ensemble *P*, tandis que la deuxième ne traite que 10 paires.

Hwang et Guo ont donné une implémentation de programmation non-dynamique efficace en temps et en espace avec $O(Ln)$ en temps et un espace d'ordre $O(n)$. Cette implémentation n'est pas limitée aux séquences d'ADN mais elle est aussi valide pour n'importe quelle séquence générale nécessitant une comparaison avec une autre séquence.

La comparaison entre certaines implémentations nous a permit de poser plusieurs questions qui ouvrent d'autres voix de recherches à fin d'adapter l'algorithme de Hwang et Guo, ou de l'étendre sur des cas plus précis dans le traitement des séquences.

Index

Complexité des algorithmes

INDEX

Complexité des algorithmes.

1 Introduction :

Pour résoudre un problème, plusieurs algorithmes sont souvent candidats (i.e. envisageables). Un choix d'algorithmes est donc nécessaire, il est souvent basé sur les coûts des algorithmes. On choisira l'algorithme le *moins coûteux* que les autres dans les conditions de réalisations égales. Il peut arriver des cas où le coût n'est pas le critère de choix, d'autres critères tels que la lisibilité, la facilité de maintenance, peuvent être utilisés.

L'intérêt de ce chapitre est de comprendre ce qu'est la complexité, les méthodes de la calculer, afin d'évaluer l'importance des algorithmes d'alignements que nous exposerons par la suite, car la plupart de ces algorithmes oeuvrent pour réaliser des programmes d'alignement à moindre coût (espace mémoire consommé et temps d'exécution).

Nous donnons ici quelques rappels élémentaires concernant la complexité algorithmique. Cette section est réalisée à partir des informations contenues [Att97], [Ber03], [Alb98].

2 Définition de la complexité : [Att97], [Alb98]

La notion de complexité des algorithmes est utilisée pour préciser l'idée selon laquelle un algorithme est caractérisé par un coût. Ce dernier se mesure essentiellement par rapport à l'utilisation des deux ressources que sont le *temps* et *l'espace mémoire*.

On parle alors de *complexité temporelle* d'un algorithme pour indiquer le nombre d'instructions nécessaires pour atteindre la solution et de *complexité spatiale* d'un algorithme pour indiquer le volume d'espace mémoire nécessaire à son fonctionnement.

Nous nous intéressons dans ce cadre au comportement de l'algorithme lorsqu'il est appliqué à des problèmes de plus en plus considérables. Généralement, lorsque l'on parle de complexité sans préciser en temps ou en espace, c'est que l'on parle de la complexité en temps.

3 Type de complexité : [Alb98][Att97]

Avant d'évaluer un algorithme en terme de complexité temporelle ou spatiale, il faut d'abord se fixer un objectif qui répond à la question : « Quelle complexité doit-on calculer ? », or on distingue, en effet, trois sortes de complexités :

- La complexité dans le pire des cas : la complexité maximale, ou complexité du *cas le plus défavorable*, qui est le temps d'exécution de l'algorithme lorsque l'on choisit les données entraînant l'exécution la plus longue.
- La complexité en moyenne : qui le temps moyen d'exécution de l'algorithme appliqué à n données quelconques. Il convient de noter que ce temps moyen n'a de sens que si l'on peut faire des hypothèses probabilistes sur la répartition des données.
- La complexité dans le meilleur des cas : ou complexité du *cas le plus favorable*, qui est le temps d'exécution de l'algorithme lorsque on choisit les données entraînant l'exécution la plus courte.

Ces notions s'étendent sans difficulté à la mesure du coût d'un algorithme en espace mémoire : on parle de *complexité spatiale* minimale, maximale ou moyenne.

On note que le type de complexité à calculer dépend de l'application (par exemple : réseau téléphonique, commande de freinage, …). Si on prend le cas d'un réseau téléphonique, on doit par exemple calculer la complexité d'un programme d'envoi de messages, au pire des cas. Ça correspond aux jours des fêtes et des grandes occasions où des millions de messages sont envoyés par les utilisateurs de ce réseau.

4 Temps de calcul : [Ber03]

On note $t(n)$ le temps de calcul d'un algorithme en fonction de la taille n des données en entrée.

Pour rester indépendant de la machine, on considère que les opérations élémentaires (additions, multiplications, comparaisons, etc.) prennent toutes le même temps. En général, on s'intéresse au temps maximum pour une valeur n donnée, c'est la complexité dans le pire des cas.

Si une opération élémentaire prend une microseconde, notées μs, voici un tableau donnant les temps de calcul suivant la taille n du problème.

$t(n)$ \ N	10	20	30	40	50	60
$\log n$	1 µs	1.3 µs	1.5 µs	1.6 µs	1.7 µs	1.8 µs
n	10 µs	20 µs	30 µs	40 µs	50 µs	60 µs
$n \log n$	10 µs	26 µs	44 µs	64 µs	85 µs	107 µs
n^2	100 µs	400 µs	900 µs	1.6 ms	2.5 ms	3.5 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
n^5	0.1s	3.2 s	24.3 s	1.7 m	5.2 m	13 m
2^n	1 ms	1 s	18 m	13 j	36 a	366siècles
3^n	59 ms	58 m	6 m	3855 siècles	$2*10^8$ siècles	$1.3*10^{13}$ siècles

Tableau 1 : rapport temps de calcul/taille du problème.

Il est important de distinguer la *complexité pratique* qui est une *mesure précise* des temps de calcul et des tailles de mémoire pour un modèle de machine bien défini, de la *complexité théorique* qui donne des *ordres de grandeur* des coûts, de façon plus indépendante des conditions pratiques de l'exécution de l'algorithme.

Comme le temps précis d'exécution dépend de l'ordinateur utilisé, on s'intéresse à son ordre de grandeur. Si le temps d'exécution d'un algorithme A est donné par $t(n) = an^2 + bn + c$, on ne considère que le premier terme de la formule, c'est-à-dire an^2, puisque les termes d'ordres inférieurs et le coefficient constant ne sont pas vraiment significatifs lorsque n est grand. On dira donc que l'algorithme A a un temps d'exécution en $\theta(n^2)$, ou que l'algorithme A a une complexité en temps en $\theta(n^2)$.

5 Principe du calcul de la complexité :

Pour arriver à calculer la complexité d'un programme, on doit adopter la démarche suivante :

1. Lister les opérations élémentaires utilisées :
   - opération arithmétique
   - accès à une variable
   - test
   - . . .
2. Déterminer la consommation de la ressource étudiée (cycles du processeur, octets de mémoire, . . .) pour chacune de ces opérations.
3. Sommer les coûts des opérations qui se déroulent séquentiellement.
4. Multiplier le coût du corps d'une boucle (resp. d'une fonction récursive) par le nombre d'exécution de la boucle (resp. d'appel de la fonction).

La valeur obtenue (complexité théorique) n'a de signification qu'à l'asymptote (but vers lequel on tend sans jamais espérer l'atteindre), on utilise usuellement les notations de Landau (***O(n²)***) pour l'exprimer.

6 Calcul pratique de la complexité : temps d'exécution : [Att97]

Considérons de nouveau une fonction $T_A(n)$ qui exprime en fonction de la taille n un certain coût maximal pour un algorithme A donné. La taille n dépend en principe du codage des données en entrée et le coût dépend des opérations effectuées dans l'algorithme.

En considérant des problèmes pour lesquels on écrit des algorithmes, on a n qui représente dans les :

- problèmes de polynômes : le degré (ou le nombre des coefficients) ;
- problèmes de matrices $m \times n$: le maximum, ou la somme ou le produit de m et n ;
- problèmes de graphes : nombre de sommets ou d'arrêtes, ou la somme des deux ;
- problèmes de tri : nombre d'éléments à trier ;
- problèmes d'analyse syntaxique : longueur du mot.

6.1 Hypothèse du coût uniforme : [Att97]

Avec l'hypothèse de *coût uniforme*, toute opération élémentaire prend un temps constant (de l'ordre de $O(1)$) ; le temps d'une opération fondamentale est proportionnelle aux coûts des opérations élémentaires qui la composent et prend lui aussi un temps constant (de l'ordre de $O(1)$).

6.2 Principales règles de comptabilisation des opérations fondamentales : [Att97]

Nous ne considérons ici que le cas des programmes séquentiels. Un programme séquentiel a une structure qui met en évidence une méthode inductive de dénombrement des opérations effectuées. Cette structure dépend des structures de contrôle utilisées.

- **La séquence**

Soit une séquence S de n actions a_1 ; a_2 ; … ; a_n.
Le coût (ou temps d'exécution) de la séquence S est :

$$\text{Coût S} = \sum_{i=1}^{n} \text{Tai} \quad \textit{où } Ta_i \text{ représente le coût de l'action } a_i.$$

- **La conditionnelle**

Soit la conditionnelle :

```
Si  C
        alors T1
        sinon T2
finsi
```

Le temps d'exécution de la conditionnelle avec $T2$ éventuellement vide est :

$$Coût\ S = coût(C) + Max\ \{coût\ (T1), coût\ (T2)\}$$

- **L'itération**

Soit l'itération :

```
Tantque C
        Faire T ;
Finfaire
```

Le temps d'exécution de la boucle se calcule de la façon suivante :

$$Coût_{tantque} = m * coût\ (C) + m * coût\ (T) + coût\ (non\ C).$$

où m est le nombre de passage dans le corps de la boucle *tantque*.

- **L'appel de procédure**

L'appel de procédure est décomposé en plusieurs opérations élémentaires. On a donc un coût uniforme qui prend un temps $O(1)$ c'est à dire majoré par une constante.

6.3 Le rapport temps d'exécution – forme des données : [Att97]

Le temps d'exécution d'un algorithme dépend non seulement de la taille des données mais aussi de la forme des données.

Pour certains algorithmes, le temps d'exécution ne dépend que de la taille des données (multiplication de matrices, opérations sur les nombres, …) ; mais dans la pratique on trouve souvent que la complexité varie aussi, pour une taille fixée des données, en fonction des données elles-mêmes.

Plusieurs quantités caractérisent le comportement d'un algorithme sur l'ensemble Dn des données de taille n. Soit $coût\ A(d)$ la complexité en temps de l'algorithme A sur la donnée d :

- Complexité dans le meilleur des cas : $Min_A(n) = min\{ coût\ _A(d) \mid d \in Dn\}$
- Complexité dans le pire des cas : $Max_A(n) = max\{ coût\ _A(d) \mid d \in Dn\}$
- Complexité en moyenne : $Moy_A(n) = \sum p(d).\ coût\ _A(d)\ ;\ d \in D_n$

où $p(d)$ est la probabilité d'avoir d en entrée de l'algorithme.

7 Outils mathématiques : [Att97]

7.1 Estimation asymptotique – Ordre de grandeur :

Dans les problèmes d'évaluation, on utilise souvent des fonctions de dénombrement. Or une estimation de l'ordre de grandeur asymptotique de la fonction fournit suffisamment d'informations. On va donc se servir de l'estimation de l'ordre de grandeur plutôt que de la fonction de dénombrement précise.

On considère uniquement les fonctions de $IN \rightarrow IR$, elles sont interprétées comme des fonctions de dénombrement. On définit ci-dessous quelques fonctions utilisées pour l'évaluation de la complexité.

Définition 1 :

On dit qu'une fonction $f : IN \rightarrow IR$ tend vers a lorsque n tend vers l'infini lorsque : $\forall \upsilon, \exists\ n_0$ tel que $n \geq n_0 \Rightarrow |f(n) - a| \leq \upsilon$

Définition 2 :

On dit que f tend vers l'infini quand n tend vers l'infini lorsque :
$\forall K, \exists\ n_0$ tel que $n \geq n_0 \Rightarrow |f(n)| \geq K$

Le fait qu'une fonction f tende vers une limite finie ou infinie quand n tend vers l'infini ne constitue pas un renseignement précis (intéressant). Pour avoir plus d'informations utiles, on compare f avec des fonctions dont on connaît le comportement au voisinage de l'infini.

Soit g une fonction positive qui ne s'annule pas au voisinage de l'infini :

Définition 3 :

On dit que f est asymptotiquement négligeable devant g et on écrit $f = o(g)$ ou $f(n)=o(g(n))$ (on prononce « petit o de g »), lorsque le rapport $\frac{f}{g}$ tend vers 0 quand $n \rightarrow \infty$.

Définition 4 :

On dit que f est asymptotiquement équivalente à g et n écrit $f \equiv g$, lorsque $\frac{f}{g}$ tend vers 1 quand $n \rightarrow \infty$.

Définition 5 :

On dit que f est asymptotiquement dominée par g et on écrit $f = O(g)$ ou $f(n)=O(g(n))$ (on prononce f égale « grand O de g ») lorsqu'il existe une constante $c > 0$ telle que, pour n assez grand, on a $|f(n)| \leq c.g(n)$

Définition 6 :

On dit que f est asymptotiquement du même ordre de grandeur que g et on écrit $f = \theta(g)$ ou $f(n) = \theta(g(n))$ (on prononce f égale thêta de g) lorsque $f = O(g)$ et $g = O(f)$.

7.2 Echelle de comparaison :

On dit qu'une fonction f est :

- *bornée ou constante* si $f = O(1)$
- *logarithmique* si $f = O(\log n)$
- *linéaire* si $f = O(n)$
- *quadratique* si $f = O(n^2)$
- *exponentielle* s'il existe une constante $c > 0$ telle que $f = O(c^n)$.

Remarque : Lorsque $f = O(g)$ ou mieux encore $f = \theta(g)$ on dit que g est un ordre de grandeur de f.

8 Classes de complexité : [11]

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

- *Les algorithmes sub-linéaires*, dont la complexité est en général en $O(\log n)$. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinalité n.
- *Les algorithmes linéaires* en complexité $O(n)$ ou en $O(n \log n)$ sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de n symboles ou les algorithmes optimaux de tri.
- Plus lents sont les algorithmes de complexité située entre $O(n^2)$ et $O(n^3)$, c'est le cas de la multiplication des matrices et du parcours dans les graphes.
- Au delà, les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

9 Conclusion

Le coût d'un algorithme en fonction des ressources (temps et mémoire) est un critère important pour comparer des algorithmes en vue d'un choix d'une solution pour un problème donné. Le coût n'est pas le seul critère utilisé dans la pratique, il ne faut pas perdre de vue les critères de simplicité de l'algorithme pour la compréhension et la mise au point.

Le coût théorique d'un algorithme permet de mesurer un algorithme indépendamment de tout critère matériel.

Le coût pratique est lié aux caractéristiques d'une machine donnée. Ce coût ne peut être comparé qu'avec d'autres coûts effectués dans les mêmes conditions.

Références

Références

[Alb98] L.Albert et al. « Cours et Exercices d'Informatique ». Ed Vuibert. 1998

[Apo87] A.Apostolico, C.Guerra. « The longest common subsequence problem revisited ». Algorithmica 2 .1987. pp 315- 336

[Att97] C.Attiogbé. « Eléments de complexité et de calculabilité »
Notes de cours – Licence. Faculté des sciences et des techniques.Université de Nantes
Juillet 1997, mars2001

[Ber03] S.Bérard. « Alignement de séquences : application à la comparaison de chaînes d'ADN »
LIRMM, UMR CNRS. Octobre 2003.

[Cro01] M.Crochemore, C.Hancart, T.Lecroq. « Algorithmique du texte ». Edition vuibert. 2001. pp 2-3.

[Guo05] J.Y. Guo, F.K.Hwang. « An almost-linear time and linear space algorithm for the longest common subsequence problem ». Information Processing Letters 94 (2005) 131–135.

[Hun77] J.W.Hunt, T. G.Szymanski. " A fast algorithm for computing longest common subsequences". Communications of ACM, volume 20, N° 5, mai 1977

[Hir75] D.S.Hirschberg, "A Linear Space Algorithm for Computing Maximal Common Subsequences". Communication of the ACM, vol 18, N°6, juin 1975

[Pev93] P.A.Pevzner, M.S.Waterman, Generalized sequence alignment and duality, Adv.Appl.Math. vol 14 (2) .1993. pp 139-171

[Smi81] T.Smith, M.Waterman, « Identification of common molecular subsequences ».Journal of Molecular Biology, N°147(1), pp 195-197

[Win99] P.C.Winter, G.I.Hickey, H.L.Fletcher. "L'essentiel en génétique". Edition Berti .1999

Références sur Internet :

[01] Université de Paris 5, « Autoformation en bioinformatique »
www.dsi.univ-paris5.fr/bio2/autof2/somm_int.htm

[02] BIONEQ, Réseau Québécois de bio-informatque
http://apps.bioneq.qc.ca/twiki/bin/view/Basedeconnaissances/ReplicationADN

[03] G.Tempesti : « BioWall » : http://lslwww.epfl.ch/pages/staff/petraglio/home.html
http://lslwww.epfl.ch/biowall/VersionF/ApplicationsF/SequenceF.html

[04] «EncycloBio : lexique de la biologie»
www.dictionnaire-biologie.com/biologie

[05] Université Joseph Fourier – Grenoble, «Algorithme d'alignement global»
http://www-fourier.ujf-grenoble.fr/~parisse/info/dynamic/node2.html

[06] E.Jaspard, «Conférence sur la bioinformatique – Introduction»
http://ead.univ-angers.fr/~jaspard/Page2/COURS/8ModuleL1CSG/

[07] Les Mathématiques sur le net «Cours de mathématiques supérieures».
www.les-mathematiques.net

[08] J.M.Richer. «Recherche en bioinformatique», Université d'Angers, France. http://www.info.univ-angers.fr/pub/richer/rec/bio/

[09] Encyclopédie du Web. http://www.webencyclo.com/les secrets de l'ADN

[10] Wikipedia« Distance de Levenshtein »
http://fr.wikipedia.org/wiki/Distance_de_Levenshtein

[11] François Morain, Cours : « algorithmes et complexité »
http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Morain/TC/X2002/Poly/www-poly008.html