Chapitre 4

Les méthodes directes de comparaison

Experiment to me
Is every one I meet
If it contain a Kernel?
The Figure of a Nut

Presents upon a Tree
Equally plausibly,
But Meat within, is requisite
To Squirrels, and to Me.

Poème #1719. Emily Dickinson.
The Complete Poems of Emily Dickinson.
Thomas H. Johnson (Ed.). Faber and Faber, 1975.

Ce chapitre a pour but de présenter l'approche directe de la comparaison des chaînes macromoléculaires. Il se trouve que cette approche englobe presque toutes les méthodes d'analyse hors celles que nous avons nous-mêmes élaborées et qui sont introduites dans le chapitre suivant. C'est donc une revue des méthodes développées par d'autres que nous donnons ici.

Ces méthodes se divisent en deux grands groupes, celles qui comparent les chaînes dans leur totalité, et celles qui effectuent des comparaisons locales. Mais nous pouvons en réaliser également une autre classification, comprenant les méthodes ayant pour base l'optimisation d'une fonction de score, les méthodes combinatoires exactes et enfin celles qui recherchent une optimisation stochastique et qui sont les plus proches des méthodes indirectes.

Le plan de ce chapitre suit ces divisions naturelles. Dans une première partie nous présentons donc les méthodes globales, puis dans une seconde celles locales. Les méthodes globales sont en général toutes basées sur des optimisations d'une certaine fonction de score. Les locales peuvent l'être, mais elles concernent également l'approche combinatoire exacte et les optimisations stochastiques. Cette seconde partie se divise ainsi en trois sous-parties.

Notre intention n'est pas de détailler chacune de ces approches. La personne intéressée peut le faire en consultant les articles cités tout au long du chapitre. Notre objectif est simplement d'essayer de dégager les idées principales qui se cachent sous une grande variété apparente de méthodes et d'introduire certaines difficultés inhérentes au problème de la comparaison multiple de chaînes.

La première de ces difficultés est d'ordre conceptuel. Nous avons vu en effet que comparer plus de deux objets à la fois est une opération pour laquelle nous ne disposons d'aucune définition 'naturelle'. Commencer par établir une telle définition est donc une tâche importante qui n'est pas toujours faite de manière très précise. Certaines définitions existent pourtant, nous les avons présentées au chapitre précédent, mais elles mènent souvent à des méthodes de calculs exacts qui sont impraticables. La seconde difficulté à laquelle nous sommes confrontés lorsque nous réalisons des comparaisons multiples est ainsi d'origine algorithmique et est essentiellement liée à la nature combinatoire du problème. Nous verrons que cela a conduit à l'élaboration de nombreuses heuristiques pour comparer un grand nombre d'objets simultanément d'une manière qui soit efficace en pratique.

4.1 Approche globale

4.1.1 Base de départ : le deux à deux

4.1.1.1 Principe — la programmation dynamique

La base de départ de toute comparaison d'un ensemble de chaînes entre elles est la comparaison de celles-ci deux à deux ainsi que nous l'avons vu au chapitre précédent. La mesure de ressemblance adoptée concerne l'optimisation d'une fonction de score associée à un alignement des deux chaînes où le score peut indiquer soit une similarité entre chaînes, soit une dissimilarité ou encore une distance.

Étant donné une telle fonction, la méthode classique et exacte pour trouver le meilleur alignement entre 2 chaînes $a_1...a_n$ et $b_1...b_m$, c'est-à-dire celui qui optimise la valeur de cette fonction, est alors la programmation dynamique [Needleman and Wunsch, 1970] [Sellers, 1974] [Wagner and Fischer, 1974]. Il ne s'agit de rien d'autre que d'une méthode algorithmique de calcul qui reflète la nature récurrente de la notion de ressemblance choisie.

Cette méthode consiste en effet à construire par induction un tableau $\mathcal{D}$ de dimension proportionnelle au produit des longueurs des 2 chaînes où la valeur de la cellule $\mathcal{D}(i, j)$ pour $1 \leq i \leq n$ et $1 \leq j \leq m$ indique le score de l'alignement optimal des chaînes jusqu'aux positions i et j respectivement et peut être déduite des valeurs de trois des cellules adjacentes du tableau, suivant en cela la définition 3.2.20. Nous avons ainsi (voir figure 4.1) :

$$\mathcal{D}(i,j) = max/min\{\mathcal{D}(i-1,j-1) + score\begin{pmatrix} a_i \\ b_j \end{pmatrix}, \mathcal{D}(i-1,j) + w_\lambda, \mathcal{D}(i,j-1) + w_\lambda\}$$

où w_λ est la valeur attribuée à un trou et la valeur de $score\begin{pmatrix} a_i \\ b_j \end{pmatrix}$ est donnée par une matrice, ou est obtenue à partir d'une relation ou d'une couverture (voir définition 3.2.19). Notons que, dans le cas d'une mesure de similarité, la valeur de w_λ est au plus égale à la plus petite valeur de substitution d'un symbole par un autre (en général, elle est donc négative).

Les valeurs frontières du tableau sont définies par :

$$\mathcal{D}(i,0) = i \times w_\lambda,\ \mathcal{D}(0,j) = j \times w_\lambda.$$

Le calcul du score optimal d'un alignement de deux chaînes peut également être vu en termes d'un parcours dans un graphe orienté dont les nœuds représentent les paires (i, j) pour $0 \leq i \leq n$ et $0 \leq j \leq m$ et où le nœud $(i-1, j-1)$ est lié par une arête à chacun des trois nœuds $(i-1, j)$, $(i, j-1)$ et (i, j) lorsque ceux-ci sont définis. Le poids des deux premières arêtes est égal à la valeur affectée à un trou et le poids de la dernière à celui de la substitution de a_i par

	–	G	G	G	T	G	A	T	T	A	G	C	T
–	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22	−24
G	−2	1	−1	−3	−5	−7	−9	−11	−13	−15	−17	−19	−21
C	−4	−1	0	−2	−4	−6	−8	−10	−12	−14	−16	−16	−18
T	−6	−3	−2	−1	−1	−3	−5	−7	−9	−11	−13	−15	−15
G	−8	−5	−2	−1	−2	0	−2	−4	−6	−8	−10	−12	−14
A	−10	−7	−4	−3	−2	−2	1	−1	−3	−5	−7	−9	−11
T	−12	−9	−6	−5	−2	−3	−1	2	0	−2	−4	−6	−8
A	−14	−11	−8	−7	−4	−3	−2	0	1	1	−1	−3	−5
T	−16	−13	−10	−9	−6	−5	−4	−1	1	0	0	−2	−2
A	−18	−15	−12	−11	−8	−7	−4	−3	−1	2	0	−1	−3
G	−20	−17	−14	−11	−10	−7	−6	−5	−3	0	3	1	−1
C	−22	−19	−16	−13	−12	−9	−8	−7	−5	−2	1	4	2
T	−24	−21	−18	−15	−12	−11	−10	−7	−6	−4	−1	2	5

Figure 4.1: Programmation dynamique — Tableau de scores ($score(a_i, b_j) = +1$ si $a_i = b_j$, -1 sinon; w_λ = -2).

b_j. Trouver le score optimal correspond alors à trouver un chemin de plus grand poids dans ce graphe (ou de plus petit dans le cas d'une mesure de dissimilarité ou d'une distance) allant du nœud source (0,0) jusqu'au nœud puits (n, m) (voir figure 4.2). Ce chemin correspond alors à celui d'un 'meilleur' alignement des deux chaînes.

Les deux approches sont souvent combinées, et un tableau de pointeurs est construit en même temps que le tableau des scores (voir figure 4.3). Ce second tableau représente en fait le graphe antérieurement donné pour lequel chaque arête se trouve maintenant orientée en sens inverse. Grâce à lui, un meilleur chemin de la source jusqu'au puits peut être retrouvé à partir du calcul du meilleur score, et ainsi un meilleur alignement de $a_1 ... a_n$ et $b_1 ... b_m$, sans avoir besoin de refaire tous les calculs à l'envers.

Lorsqu'une des chaînes est beaucoup plus courte que l'autre, il est clair que c'est uniquement le meilleur alignement de cette chaîne plus courte avec un mot de l'autre qui est recherché. La formule de récurrence demeure la même, seules changent les valeurs des cellules situées à une des frontières du tableau. En plaçant la chaîne plus longue en haut de celui-ci, on a en effet $D(i, 0) = 0$ pour $1 \leq i \leq n$. L'alignement optimal est alors obtenu en partant d'une cellule de la dernière ligne du tableau ayant la valeur maximale (ou minimale) et en remontant le tableau des pointeurs jusqu'à la première ligne. Cette variante consiste à réaliser un meilleur ajustement de la chaîne courte avec celle plus longue (en anglais, 'Best fit').

Observons enfin que l'alignement optimal ne représente pas toujours celui qui est 'correct' biologiquement, ce dernier correspondant plutôt à un alignement ayant un score élevé proche mais non égal au maximum (minimum). Des variantes de la programmation dynamique existent alors qui calculent tous les alignements globaux dont le score se trouve à une certaine distance du score optimal [Waterman, 1983] [Waterman and Byers, 1985] [Zuker, 1991].

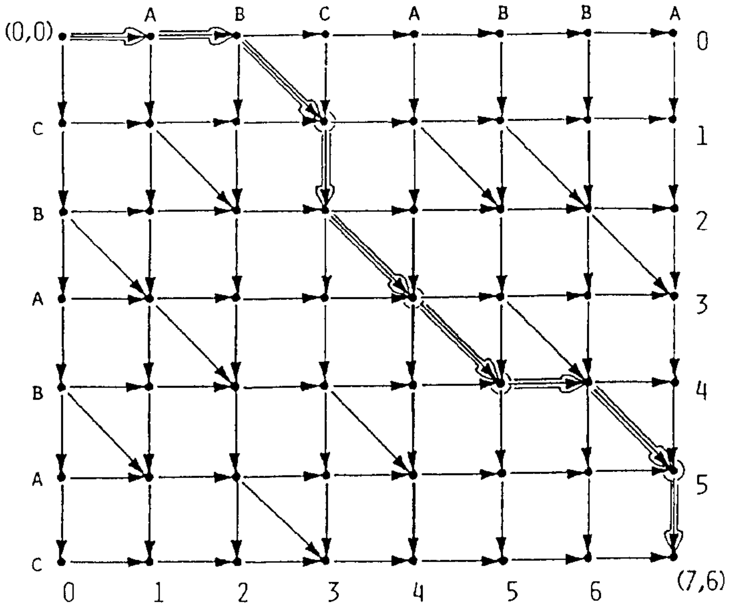


Figure 4.2: Programmation dynamique — Graphe.

D(i,j)			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7
v	1	↑ 1	↖ 1	↖ ← 2	↖ ← 3	↖ ← 4	↖ ← 5	↖ ← 6	↖ ← 7
i	2	↑ 2	↖ ↑ 2	↖ 2	↖ 2	← 3	← 4	← 5	← 6
n	3	↑ 3	↖ ↑ 3	↖ ↑ 3	↖ ↑ 3	↖ 3	↖ ← 4	↖ ← 5	↖ ← 6
t	4	↑ 4	↖ ↑ 4	↖ ↑ 4	↖ ↑ 4	↖ 3	↖ ← 4	↖ ← 5	↖ ← 6
n	5	↑ 5	↖↑ 5	↖↑ 5	↖ ↑ 5	↑ 4	↖ 4	↖ ← 5	↖ ← 6
e	6	↑ 6	↖ ↑ 6	↖ ↑ 6	↖ ↑ 6	↑ 5	↖ 4	↖ ← 5	↖ ← 6
r	7	↑ 7	↖ ↑ 7	↖ 6	↖ ← ↑ 7	↑ 6	↑ 5	↖ 4	← 5

Figure 4.3: Programmation dynamique — Tableau des scores et tableau de pointeurs ($score(a_i, b_j) = 0$ si $a_i = b_j$, $+1$ sinon; $w_\lambda = +1$).

4.1.1.2 Une discussion sur la complexité

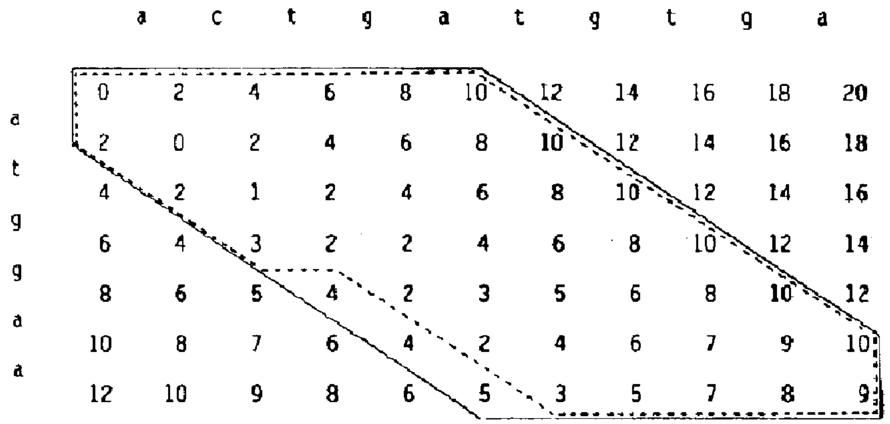
4.1.1.2.1 Temps

La complexité en temps de l'algorithme de programmation dynamique est $O(n.m)$. Il a été montré [Wong and Chandra, 1976] que ce temps est aussi $\Omega(n.m)$ pour des alphabets de taille infinie, lorsque seules des comparaisons du type 'égal-ou-différent' peuvent être réalisées et que la complexité de l'algorithme est caractérisée exclusivement en termes de la taille de l'entrée. Cette borne inférieure est obtenue même lorsque le système de score mesure un cas spécial de la distance d'édition entre les chaînes [Aho *et al.*, 1974] où seules les délétions et insertions sont autorisées. Ce que nous recherchons alors est une plus longue sous-séquence commune à celles-ci ($s' = c_1 ... c_p$ est appelée une sous-séquence d'une chaîne $s = a_1 ... a_n$ si s' peut être obtenu de s en supprimant certains symboles de s).

Lorsque l'alphabet est de taille finie, et que les scores attribués aux paires représentent une distance d'édition ou ont des poids qui sont des entiers multiples d'un même nombre réel r, il est possible, en utilisant la technique appelée des Quatre Russes, de calculer le score d'un alignement global des deux chaînes $s1 = a_1 ... a_n$ et $s2 = b_1 ... b_m$ en un temps $O((n.m)/\log n)$ [Masek and Paterson, 1983] si l'on suppose, sans perte de généralité, que $n \leq m$.

La complexité en temps de l'algorithme de programmation dynamique peut être également caractérisée en termes d'autres paramètres que la seule taille de l'entrée. Lorsque nous cherchons par exemple à déterminer la distance d'édition entre $s1$ et $s2$, un paramètre d peut être défini qui représente une borne supérieure estimée de cette distance. Des variantes de l'algorithme de programmation dynamique ont alors été élaborées [Fickett, 1984] [Nakatsu *et al.*, 1982] [Spouge, 1989] [Spouge, 1991] [Ukkonen, 1985a] [Ukkonen, 1985b] qui calculent le score optimal global des deux chaînes $s1$ et $s2$ en un temps $O(n.d)$. La technique consiste à limiter le volume de l'espace à explorer, c'est-à-dire le nombre de cellules du tableau $\mathcal{D}$ qu'il faut effectivement calculer (ou de nœuds du graphe par lesquels il faut passer). L'idée est la suivante. Rappelons que nous raisonnons dans cet exemple en termes de distance plutôt que de similarité, et utilisons la double représentation tableau de scores-graphe (i.e. tableau de pointeurs). Lorsque les deux chaînes $s1$ et $s2$ ne sont pas trop dissemblables, un chemin optimal dans le graphe se situe en général proche de la diagonale principale du tableau, en fait il est entièrement contenu dans une bande de largeur $2d$ centrée sur cette diagonale (voir figure 4.4). Seules les cellules situées à l'intérieur de cette bande ont alors besoin d'être calculées.

Il est possible d'être encore plus économe que cela en travaillant non avec une seule estimation — celle du paramètre d — mais avec deux. La seconde représente non plus une borne supérieure de la distance $\mathcal{D}(n, m)$ entre $a_1 ... a_n$ et $b_1 ... b_m$ mais une borne inférieure κ de cette distance [Spouge, 1989] [Spouge, 1991]. Cette borne est en fait estimée pour chaque nœud x du graphe, et indique la valeur minimale que peut avoir la longueur d'un chemin passant par $x = (i, j)$ et allant de la source au puits. Elle est donc notée $\kappa_x(\text{source,puits}) = \kappa(\text{source,}x)$ + $\kappa(x\text{,puits})$. Elle permet ainsi d'éventuellement réduire encore plus le nombre de cellules à calculer à l'intérieur de la bande, cela à la fois de manière directe (si pour un x donné à l'intérieur de celle-ci on a $\kappa_x(\text{source,puits}) > d$), et de manière indirecte puisque l'estimation de $\kappa_x(\text{source,puits})$ pour tout x permet d'établir un ordre parmi les cellules du tableau. Celles ayant une plus petite valeur estimée de κ_x sont alors calculées en priorité et la cellule $\mathcal{D}(n, m)$ peut être atteinte avant d'avoir eu à déterminer la valeur des autres cellules situées dans la bande de largeur $2d$. Aussi bien d que κ_x peuvent être estimés de façon statique, c'est-à-dire une fois pour toutes, ou dynamique. Dans ce second cas et en ce qui concerne κ_x, on a $\kappa_x(\text{source,puits})$ = $\mathcal{D}(i, j)$ + $\kappa(x\text{,puits})$ (rappelons que (i, j) représente le nœud x et $\mathcal{D}(i, j)$ la distance entre $a_1 ... a_i$ et $b_1 ... b_j$). Une estimation dynamique de la valeur de d dépend elle aussi du chemin

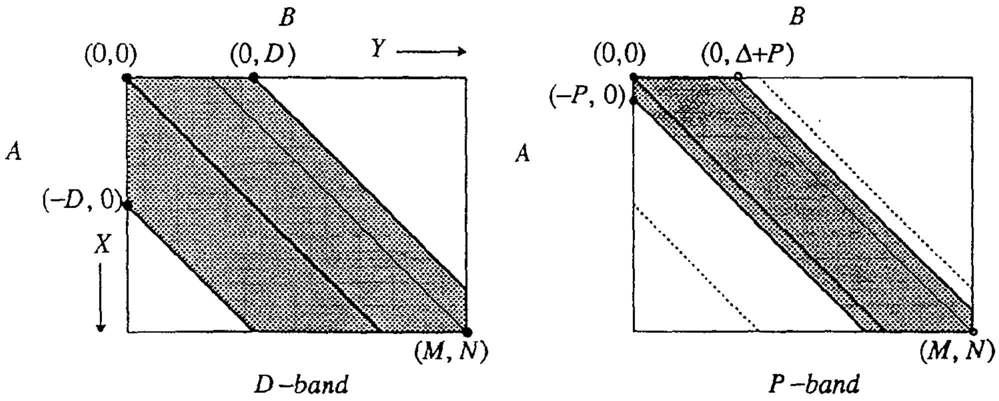


		a	c	t	g	a	t	g	t	g	a
	0	2	4	6	8	10	12	14	16	18	20
a	2	0	2	4	6	8	10	12	14	16	18
t	4	2	1	2	4	6	8	10	12	14	16
g	6	4	3	2	2	4	6	8	10	12	14
g	8	6	5	4	2	3	5	6	8	10	12
a	10	8	7	6	4	2	4	6	7	9	10
a	12	10	9	8	6	5	3	5	7	8	9

Figure 4.4: Bande $2d$ à l'intérieur de laquelle se situent les chemins optimaux — algorithme de Fickett et Ukonnen (bande plus étroite) ($score(a_i, b_j) = 0$ si $a_i = b_j$, $+1$ sinon; $w_\lambda = +2$).

déjà parcouru et on a alors d_x(source,puits) $= \mathcal{D}(i,j) + \mu(x,$puits$)$ où $\mu(x,$puits$)$ est une borne supérieure pour la distance séparant $a_{i+1}...a_n$ et $b_{j+1}...b_m$. L'estimation dynamique des bornes inférieures et supérieures de la distance $\mathcal{D}(n,m)$ permet bien sûr de mieux réduire l'espace à explorer mais elle représente elle-même un certain coût en espace et en temps. Ce coût est en général d'autant plus élevé que l'estimation est bonne et un équilibre entre réduire l'espace au mieux et dépenser trop de temps à estimer les bornes doit donc être trouvé. La différence essentielle entre les versions mentionnées ci-dessus porte justement sur la manière de réaliser cette estimation de d et κ. Ficket par exemple n'estime que la valeur de d et le fait de manière statique. De même qu'Ukkonen, il 'devine' en fait cette valeur, et éventuellement la revoit à la hausse si dans une première application de l'algorithme la cellule (n,m) du tableau n'est pas atteinte (parce que $\mathcal{D}(n,m)$ est supérieur à la valeur 'devinée'). Spouge estime de manière dynamique aussi bien la valeur de d_x(source,puits) que celle de κ_x(source,puits) pour tout nœud $x = (i,j)$ du graphe d'après le nombre maximal de substitutions, délétions et insertions (estimation de d_x) ou le nombre minimal de délétions et insertions (estimation de κ_x) nécessaires pour aligner $a_{i+1}...a_n$ et $b_{j+1}...b_m$.

Une réduction de l'espace à explorer peut être obtenue avec des mesures de ressemblance entre chaînes autres que la distance d'édition. Celles-ci peuvent ainsi concerner des mesures de similarité ou de dissimilarité quelconques si le poids attribué à un trou est constant, ou bien une forme spéciale de la distance d'édition déjà mentionnée où seules des délétions et insertions sont autorisées entre $a_1...a_n$ et $b_1...b_m$, c'est-à-dire lorsque l'on recherche en fait une sous-séquence maximale commune aux deux chaînes. Dans ce dernier cas, Myers obtient un algorithme initialement en $O(n.d)$ où d mesure cette fois la taille de la plus petite suite d'opérations de délétions et insertions nécessaires pour passer de $s1 = a_1...a_n$ à $s2 = b_1...b_m$ [Myers, 1986], puis en $O(n.p)$ où p est le nombre de délétions dans cette suite et est égal à $\frac{1}{2}d - \frac{1}{2}(m-n)$ [Wu *et al.*, 1990]. Comme précédemment pour Spouge, on ne parcourt pas la matrice de manière strictement linéaire ligne par ligne et colonne par colonne, mais on serpente à l'intérieur d'une bande de largeur $2d$ ou $2p + (n-m)$, l'algorithme calculant en



D-band and *P-band* of an edit graph.

		a	c	e	b	d	a	b	b	a	b	e
	0	1	2	3	4	5	6					
a	1	0	1	2	3	4	5	6				
c	2	1	0	1	2	3	4	5	6			
b	3	2	1	2	1	2	3	4	5	6		
d	4	3	2	3	2	1	2	3	4	5	6	
e	5	4	3	4	3	2	3	4	5	6		
a	6	5	4	3	4	3	2	3	4	5	6	
c		6	5	4	5	4	3	4	5	6		
b			6	5	6	5	4	5	6		6	
e				6		6	5	6				6
d							6					

D −band

		a	c	e	b	d	a	b	b	a	b	e	d
	0	0	0	1	2								
a	1	0	0	0	1	2							
c	2	1	0	0	0	1	2						
b		2	1	1	0	0	1	2					
d			2	2	1	0	0	1	2				
e				2	2	1	1	1	2				
a						2	1	1	1	2			
c							2	2	2	2			
b								2	2	2	2		
e												2	
d													2

P −band

Figure 4.5: Bande $2d$ ou $2p + (n - m)$ à l'intérieur de laquelle se situent les chemins optimaux — Algorithme de Myers.

effet successivement les cellules de la bande jusqu'à une certaine frontière correspondant à un nombre de trous (ou de délétions) allant de 0 à d (ou p) (voir la figure 4.5).

Toutes ces économies qui peuvent être réalisées valent aussi bien pour le calcul du seul score d'un alignement optimal que pour l'obtention d'un tel alignement. Il est très important d'observer toutefois que ces économies dépendent des données. Dans la pire des hypothèses, c'est-à-dire en termes d'une distance d'édition si $d = m$ (les chaînes sont totalement 'différentes'), la complexité demeure en $O(n.m)$. Signalons que toutes ces techniques d'exploration exhaustives avec bornes correspondent à ce qui est appelé en anglais du 'branch-and-bound'.

Enfin, toutes les complexités présentées jusqu'à maintenant ne concernent que le cas où le score attribué à un trou est une fonction constante ou linéaire de sa longueur. Lorsqu'une fonction linéaire par t morceaux est utilisée, la complexité devient $O(n^2.t)$ si l'on suppose que $m = n$ [Gotoh, 1982] [Miller and Myers, 1988], elle est en $O(n^2. \log n)$ pour une fonction concave [Galil and Giancarlo, 1989] [Miller and Myers, 1988] [Waterman, 1984a] et en $O(n^3)$ pour une fonction quelconque [Waterman *et al.*, 1976].

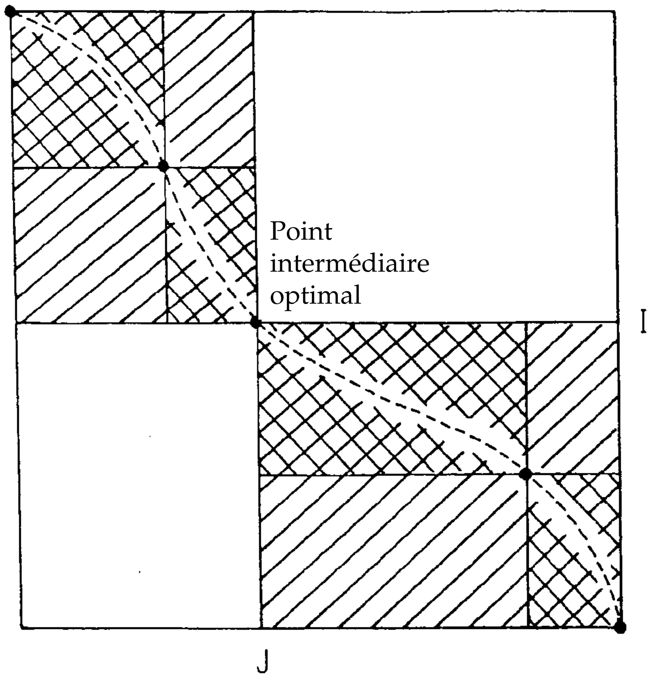


Figure 4.6: Programmation dynamique en $O(n)$ espace.

4.1.1.2.2 Espace

La complexité en espace de l'algorithme de programmation dynamique pour le cas global est en $O(min(n, m))$ aussi bien pour le calcul du meilleur score que pour la détermination d'un meilleur alignement (chemin dans le graphe).

En ce qui concerne le calcul du meilleur score, il est en effet assez intuitif de voir que nous n'avons besoin de connaître que les valeurs des cellules de la ligne $i-1$ (colonne $j-1$) pour calculer celles de la ligne i (colonne j). Il est moins facile de comprendre pourquoi trouver un meilleur chemin peut également être réalisé en $O(min(n, m))$ espace. L'idée est de diviser successivement le problème initial en deux sous-problèmes de taille plus petite impliquant l'alignement pour tout j et pour $n \leq m$ de $a_1...a_{\frac{n}{2}}$ avec $b_1...b_j$ et de $a_{\frac{n}{2}+1}...a_n$ avec $b_{j+1}...b_m$ et pour chaque sous-problème de déterminer un point intermédiaire j_{opt} optimal [Myers and Miller, 1988]. La liste de ces j_{opt} constitue alors un chemin optimal recherché (voir figure 4.6).

4.1.1.2.3 Commentaire

Si nous nous sommes attardés sur cette question de la complexité de l'algorithme de programmation dynamique c'est qu'elle va devenir cruciale lorsque nous passerons dès la section suivante au cas d'un alignement ou d'une comparaison multiple de chaînes. Nous allons voir en effet qu'une extension directe de la programmation dynamique au cas de plus de deux chaînes résulte en un algorithme qui est exponentiel dans le nombre de celles-ci qu'il y a à comparer. Ce qu'il est important à noter est que ce phénomène est intrinsèque au problème lorsque celui-ci est traité dans sa plus grande généralité et il a été montré que, dans ce cas, ce problème est en fait NP-complet [Garey and Johnson, 1979]. Les gains qui peuvent être obtenus malgré tout vont donc toujours dépendre de la nature particulière des données à traiter.

Nous verrons plus tard que le cas des comparaisons locales est différent. Une application directe de la programmation dynamique conduit à une explosion combinatoire des opérations à effectuer qui est pire en pratique que dans le cas global dans la mesure où plus aucune économie n'est possible. Cependant des méthodes indirectes de comparaison qui sont également récursives dans leur principe permettent, sous certaines conditions, d'obtenir des temps théoriques de calcul bien moindres tout en demeurant exacts et exhaustifs.

4.1.2 Le multiple exact

4.1.2.1 Du deux à deux au multiple : rappel sur les scores

Commençons par rappeler que les scores d'un alignement multiple peuvent être calculés essentiellement de deux façons. Dans la première, le score d'un alignement de plus de deux chaînes est la somme des scores de tous les alignements deux à deux qu'il induit (voir la figure 3.19). On parle dans ce cas de scores SP (voir section 3.4.1.2). La seconde utilise un arbre phylogénétique pour obtenir un alignement des chaînes et le score de cet alignement est la somme de tous les alignements deux à deux associés à chaque branche de l'arbre et qui sont induits par l'alignement multiple (voir la figure 3.21). En général, l'arbre est connu au départ (il est supposé avoir été déterminé par d'autres moyens qu'une analyse des chaînes macromoléculaires) et il s'agit simplement de trouver les chaînes ancestrales situées aux nœuds internes de l'arbre (les chaînes à aligner étant placées aux feuilles) (voir section 3.4.1.3). Des algorithmes exacts n'existent que dans le cas où l'arbre est donné *a priori* et nous ne parlerons pas du tout du cas où celui-ci doit être construit en même temps que l'alignement [Vingron and von Haesler, 1994].

4.1.2.2 Algorithmes exacts utilisant des scores SP

4.1.2.2.1 Extension directe de la programmation dynamique — problème de la complexité

Les algorithmes permettant de construire des alignements globaux optimaux sur la base de scores SP sont des extensions directes de la programmation dynamique à plus de deux dimensions. Ils ont été élaborés par Murata [Murata, 1990] [Murata *et al.*, 1985] et Gotoh [Gotoh, 1986] et consistent à calculer par induction les valeurs d'un tableau $\mathcal{D}$ à N dimensions et non plus à deux, où N est le nombre de chaînes à comparer ou aligner et $\mathcal{D}(i_1, ..., i_N)$ indique le score d'un alignement optimal des chaînes $s_1, ..., s_N$ jusqu'aux positions $i_1, ..., i_N$ respectivement. La valeur de la cellule $(i_1, ..., i_N)$ du tableau $\mathcal{D}$ peut être obtenue, de la même façon que pour le deux à deux, directement des valeurs des $2^N - 1$ cellules adjacentes (voir figure 4.7 pour $N = 3$). À nouveau ici, le problème peut également être vu comme celui de trouver un chemin de plus grand (ou de plus petit) poids dans un graphe orienté.

La principale difficulté avec cette approche est, ainsi que nous l'avons mentionné, la complexité en temps et en espace d'un tel algorithme. Celle-ci est en effet en $O(n^N)$ où n est la longueur moyenne des chaînes ($O(n^{N-1})$ espace lorsque seul le meilleur score est recherché). Il est clair que la méthode n'est envisageable que pour des valeurs petites de N, en pratique elle est en fait limitée à au plus 3 chaînes.

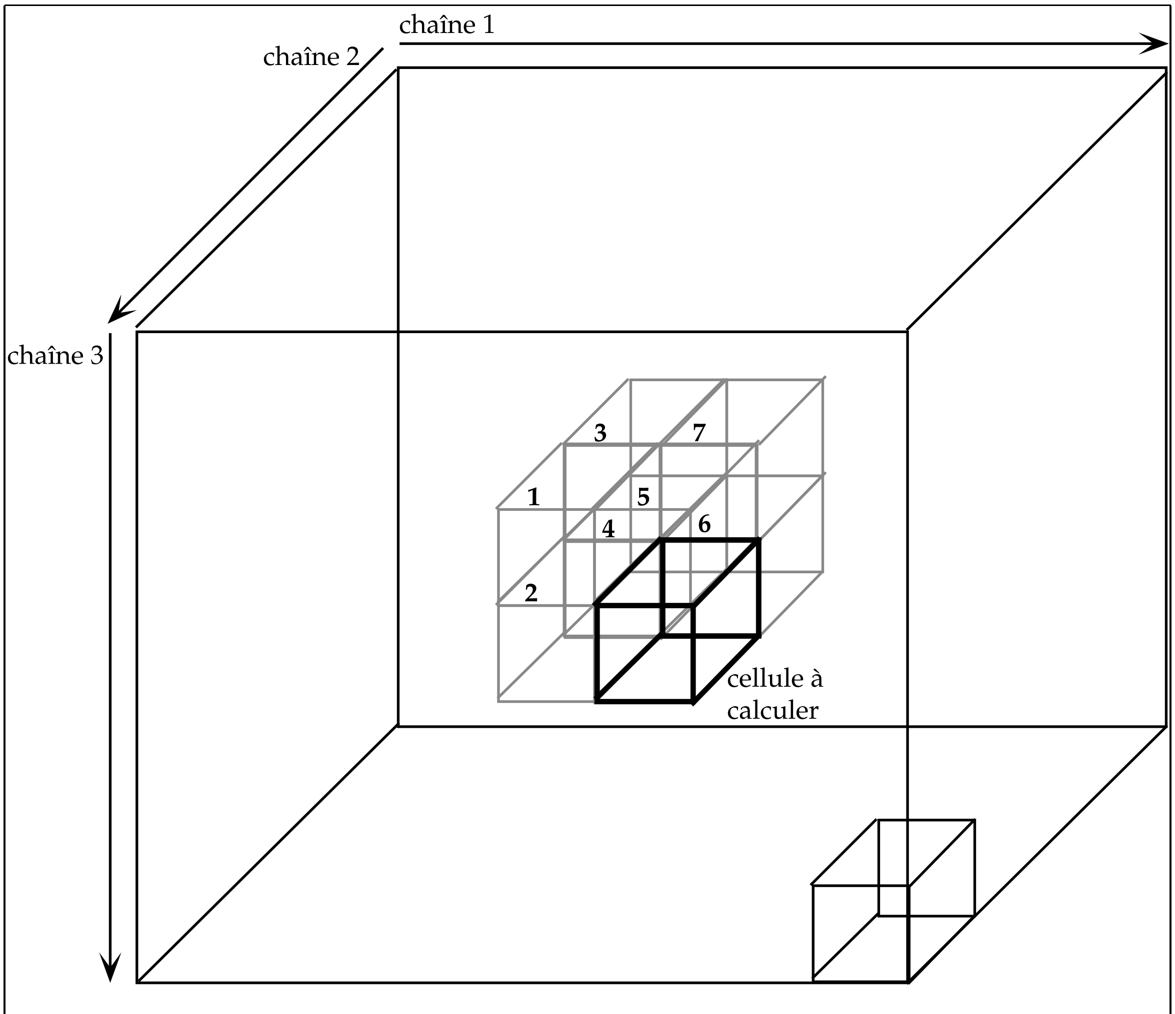


Figure 4.7: Programmation dynamique en 3 dimensions.

4.1.2.2.2 Économies possibles

Comme dans le cas du deux à deux, il est cependant possible d'essayer de limiter le volume de l'espace à explorer [Carrillo and Lipman, 1988] [Gupta *et al.*, 1995] [Lipman *et al.*, 1989], c'est-à-dire le nombre de cellules du tableau à N dimensions qu'il faut calculer. L'idée de base est la même que précédemment dans la mesure où il s'agit d'estimer certaines bornes, inférieures ou supérieures, du score recherché, mais elle exploite cette fois la nature des scores utilisés pour mesurer la qualité d'un alignement multiple. Comme auparavant, raisonnons en termes de distance plutôt que d'une mesure de similarité. Supposons qu'un alignement multiple quelconque de N chaînes ait pu être déterminé, par une heuristique par exemple. Notons son score $score(\mathcal{H})$. Si $score(\mathcal{O})$ est le score d'un alignement optimal des N chaînes, alors nécessairement $score(\mathcal{H}) \geq score(\mathcal{O})$. Mais par définition des scores SP, $score(\mathcal{O})$ est lui-même égal à la somme des scores de toutes les projections de l'alignement multiple $\mathcal{O}$ sur les N chaînes prises deux à deux, c'est-à-dire :

$$score(\mathcal{O}) = \sum_{1 \leq i < j \leq N} score(\mathcal{O}_{ij})$$

où $\mathcal{O}_{ij}$ dénote la projection de l'alignement $\mathcal{O}$ sur les chaînes si et sj. Par ailleurs, chacun de ces scores entre paires de chaînes si, sj est supérieur ou égal à la distance entre si et sj. On a donc :

$$score(\mathcal{H}) \geq score(\mathcal{O}) = \sum_{1 \leq i < j \leq N} score(\mathcal{O}_{ij}) \geq \sum_{1 \leq i < j \leq N} distance(si, sj)$$

où $distance(si, sj)$ mesure la distance entre si et sj. En distinguant l'une de ces paires, on obtient :

$$\sum_{\substack{1 \leq i < j \leq N \\ i \neq p, j \neq q}} score(\mathcal{O}_{ij}) \geq score(\mathcal{O}_{pq}) - distance(sp, sq) \; + \sum_{1 \leq i < j \leq N} distance(si, sj).$$

Ceci implique que :

$$score(\mathcal{O}_{pq}) \leq score(\mathcal{H}) + distance(sp, sq) - \sum_{1 \leq i < j \leq N} distance(si, sj).$$

En d'autres termes, le score de la projection d'un alignement optimal $\mathcal{O}$ de N chaînes sur une paire donnée de celles-ci est inférieur ou égal au score d'un alignement quelconque $\mathcal{H}$ des N chaînes moins la somme des distances entre toutes les paires restantes. En pré-calculant toutes ces distances deux à deux et en déterminant le score d'un alignement quelconque $\mathcal{H}$ des N chaînes, il est ainsi possible d'obtenir des bornes supérieures pour les scores des projections d'un alignement optimal sur chacune des paires de chaînes de l'ensemble de départ. De telles bornes limitent le nombre de cellules qu'il faut calculer sur chaque tableau projeté à deux dimensions, et donc limite le nombre de cellules qu'il faut calculer dans le tableau à N dimensions.

La différence fondamentale entre les algorithmes de Carrillo et Lipman d'un côté et de Lipman *et al.* et Gupta *et al.* d'un autre est que ces derniers peuvent dériver les bornes pour les scores des paires directement de l'alignement $\mathcal{H}$ et non de l'inégalité donnée ci-dessus. Cette borne est alors, pour une paire de chaînes donnée sp et sq, non pas le score de $\mathcal{H}$ moins la somme des distances entre toutes les autres paires, mais directement le score de la projection de $\mathcal{H}$ sur sp et sq. Ce score peut être plus petit que la borne estimée par Carrillo et peut ainsi mieux réduire l'espace à explorer, mais il ne garantit pas qu'un score optimal sera obtenu à la fin. Par ailleurs, Gupta *et al.* calculent aussi une borne inférieure aux scores des paires comme il était fait dans le cas du deux à deux (estimation de κ_x). Enfin, les algorithmes de Altschul et de

Gupta différent également de celui de Carrillo dans la mesure où les deux premiers pondèrent la contribution de chaque chaîne à l'alignement final. L'idée est de compenser l'influence trop forte que pourrait avoir la présence dans un alignement de plusieurs chaînes presque identiques.

Quel que soit l'algorithme utilisé parmi ceux mentionnés ici, la complexité de pire cas est la même que celle d'une extension pure et simple de la programmation dynamique, c'est-à-dire $O(n^N)$. En pratique cependant, les méthodes de cette section permettent d'aligner de 6 à 8 chaînes en un temps 'raisonnable'.

4.1.2.2.3 Une nouvelle approche plus orientée graphe

Signalons une autre approche, également exacte et NP-complète, de l'alignement d'un ensemble de chaînes qui est formalisée de manière différente, faisant plus exclusivement appel à la théorie des graphes (recherche d'un chemin mais aussi calcul de flots), et qui a été proposée par Kececioglu [Kececioglu, 1993]. Le problème d'un alignement multiple de chaînes est dans ce cas modélisé par celui d'une trace de poids maximal dans un graphe d'alignement. Un tel graphe est composé d'un ensemble N de nœuds représentant les symboles dans les chaînes, d'un ensemble A d'arêtes liant deux nœuds si les symboles correspondant à ceux-ci se situent sur des chaînes différentes (quelles que soient leurs positions) et sont identiques, et enfin d'un ordre $\prec$ sur les nœuds où $v \prec w$ pour $v, w \in N$ si et seulement si le symbole v vient immédiatement avant le symbole w dans une des chaînes. Un sous-graphe connexe dans un tel graphe représente une colonne de symboles identiques et donc une colonne dans un alignement possible des chaînes. Un ensemble de tels sous-graphes connexes forme alors un alignement valable de ces chaînes, appelé une trace, s'ils peuvent être ordonnés de façon à respecter l'ordre $\prec$ symbole par symbole. Pour vérifier cela, l'ordre $\prec$ est en fait étendu à un ordre $\prec^*$ sur les sous-graphes connexes où $c_1 \prec^* c_2$ pour c_1 et c_2 des sous-graphes connexes du graphe s'il existe au moins une paire de symboles v et w dans c_1 et c_2 respectivement tels que $v \prec w$. Une trace dans un tel graphe est dans ce cas défini comme étant un sous-ensemble T de A tel que la relation $\prec^*$ sur les composantes connexes de T est acyclique. Rechercher une trace de poids maximal correspond à rechercher un alignement optimal sur la base de scores SP et est effectuée par une technique d'exploration exhaustive avec bornes. En pratique, l'algorithme permet d'aligner jusqu'à 6 chaînes de longueur environ 250 symboles en quelques minutes, sa performance en moyenne est donc similaire aux méthodes de programmation dynamique plus classiques. Cette approche peut se révéler par contre plus intéressante que les précédentes si les symboles sont remplacés par des mots communs identiques ou similaires et si la relation $\prec^*$ devient une relation sur ces mots.

4.1.2.3 Algorithmes exacts utilisant un arbre phylogénétique

4.1.2.3.1 Application directe de la récurrence — problème de la complexité

Les algorithmes exacts utilisant un arbre phylogénétique qui ont été élaborés à ce jour supposent, ainsi que nous l'avons vu, que cet arbre soit connu à l'avance. Cet arbre est composé de nœuds externes correspondant aux feuilles et où sont placées les chaînes de l'ensemble de départ et de un ou plus de nœuds internes associés à des chaînes ancestrales initialement inconnues. Un arbre avec un seul nœud interne est appelé une étoile [Altschul and Lipman, 1989] (voir la section 3.4.1.5). Ce qui est recherché est un alignement optimal des chaînes de départ, c'est-à-dire un alignement correspondant à un nombre minimal de changements (substitutions, délétions et insertions) par rapport à toutes les chaînes, celles de départ plus les chaînes ancestrales.

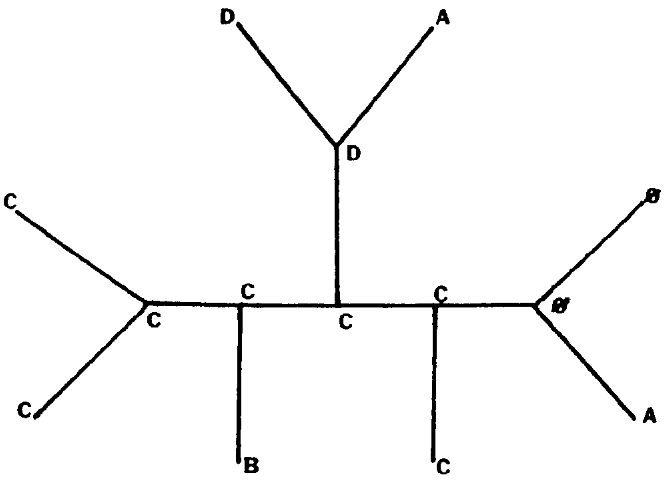


Figure 4.8: Programmation dynamique utilisant un arbre phylogénétique — exemple extrait du livre de Sankoff et Kruskal [Sankoff and Kruskal, 1983].

Le premier algorithme pour résoudre ce problème a été conçu par Sankoff [Sankoff, 1975]. Le processus de construction de l'alignement est inductif comme pour l'alignement de plusieurs chaînes utilisant des scores SP. Le score de l'alignement optimal de N chaînes jusqu'aux positions $i_1, ..., i_N$ est ainsi égal au minimum du score de l'alignement optimal obtenu un cran avant, c'est-à-dire, jusqu'aux positions i_j ou $i_j - 1$, plus une minimisation interne qui représente la plus petite distance d'édition pour, étant donné une certaine attribution de symboles des chaînes de départ ou de trous aux feuilles, toute attribution possible de symboles aux nœuds internes, y compris des trous (voir la page 86 ainsi que l'illustration de la figure 4.8).

Si $\mathcal{O}$ est un alignement optimal des N chaînes, on obtient ainsi à la fin

$$score(\mathcal{O}) = \sum_{a \in A} score(\mathcal{O}_a)$$

où A est l'ensemble des arêtes de l'arbre et $score(\mathcal{O}_a)$ est le score de la projection de $\mathcal{O}$ sur l'arête a.

L'algorithme de Sankoff est en $O((2n)^N M)$ où n et N sont comme avant la longueur moyenne et le nombre des chaînes respectivement et M est le nombre des chaînes ancestrales (correspondant au nombre de nœuds internes de l'arbre donné au départ).

4.1.2.3.2 Économies possibles

Comme dans le cas des scores SP, il est possible d'essayer d'économiser sur le nombre d'opérations à réaliser en estimant des bornes supérieures aux scores des projections d'un alignement $\mathcal{H}$ obtenu par une heuristique sur chacune des paires de chaînes de l'ensemble de départ [Altschul and Lipman, 1989]. Le score de la projection sur chaque paire (si, sj) est cependant donnée désormais uniquement par le coût de la projection du score de $\mathcal{H}$ sur chacune des arêtes de l'arbre reliant si à sj. Pour obtenir une estimation de la borne supérieure au score d'une telle projection de $\mathcal{H}$, on va comme auparavant soustraire du score de $\mathcal{H}$ la somme des distances entre toutes les paires (sk, sl) autres que (si, sj) puisque chacune de ces distances représente une borne inférieure à la distance réelle, mais imposée par l'arbre, entre sk et sl. Il va y avoir cependant une importante différence avec le cas précédent. En effet, certains chemins entre paires quelconques de chaînes peuvent posséder des arêtes en commun et il faut s'assurer que le coût de celles-ci n'est retiré du score de $\mathcal{H}$ qu'une seule fois. À ce détail près, le calcul des bornes supérieures pour la projection de $\mathcal{H}$ sur chaque paire de chaînes de l'ensemble de départ permet, comme pour les scores SP, de diminuer le nombre de cellules dont il faut déterminer la valeur dans le tableau de programmation dynamique.

4.1.3 Les heuristiques

4.1.3.1 Deux catégories d'heuristiques

Tous les algorithmes de comparaison ou alignement multiple que nous venons de voir sont exponentiels dans le nombre de chaînes à analyser. Ainsi que nous l'avons fait observer, le problème dans sa plus grande généralité est NP-complet et les astuces permettant d'effectuer moins de calculs dépendent fortement de la nature des données à traiter.

Il s'est avéré nécessaire par conséquent d'élaborer des heuristiques pour comparer globalement un ensemble de chaînes dans le but de les aligner. L'appel à une heuristique introduit toujours la nécessité d'un choix. Celui-ci peut s'appuyer sur un critère purement algorithmique (ne pas explorer tout l'espace de recherche) ou purement biologique (ne pas prendre les chemins dont on peut supposer dès le départ qu'ils conduiront à des résultats qui ne sont pas intéressants). En général les deux critères sont mélangés et la division en deux catégories que nous présentons ici, bien que paraissant suivre la ligne esquissée plus haut, ne lui est pas tout à fait fidèle. Un choix effectué sur la base de critères purement algorithmiques peut ainsi être biologiquement tout à fait justifiable, et vice-versa, un choix réalisé pour des motifs biologiques peut conduire à une manière de réduire l'espace à explorer qui est clairement exprimable en termes mathématiques.

La première catégorie d'heuristiques que nous présentons concerne alors celles qui demeurent proches des algorithmes exacts dans le sens où elles adoptent les mêmes définitions de la ressemblance que celles-ci. Elles travaillent ainsi d'une part avec des scores SP ou des scores obtenus sur la base d'un arbre phylogénétique et, d'autre part, elles cherchent à s'approcher au mieux du résultat obtenu par les algorithmes d'optimisation d'une fonction de score. Parmi celles qui ont été proposées, deux sont particulièrement intéressantes dans la mesure où justement elles garantissent sinon l'optimalité du résultat obtenu, du moins une certaine 'qualité' à ce résultat où par qualité on entend, une distance maximale du résultat considéré optimal par rapport aux définitions de ressemblance adoptées. Les économies que toutes réussissent à réaliser résultent du fait que, soit elles réduisent le nombre de chaînes à comparer (c'est le cas de l'heuristique utilisant le concept d'étoile centrale), soit elles placent une contrainte sur le type d'arbre phylogénétique autorisé.

Les heuristiques de la seconde catégorie s'éloignent quant à elles de plus en plus des algorithmes exacts, cela de deux façons. La première concerne la mesure de similarité multiple utilisée. Le score d'un alignement de plus de deux chaînes représente ainsi rarement la somme des scores de tous les alignements deux à deux induits, ou la somme des scores des alignements deux à deux suivant un arbre. La seconde différence avec les méthodes exactes est liée à la première d'une certaine manière puisque n'ayant plus de fonction de score nettement établie, il n'est plus question d'optimiser globalement l'alignement multiple. La plupart de ces heuristiques continue cependant d'optimiser le score de tous les alignements deux à deux. Ceux-ci peuvent concerner non seulement deux chaînes mais faire référence aussi à une chaîne et à un alignement ou à deux alignements préalablement obtenus. Il est important d'observer que les plus soigneusement conçues parmi ces heuristiques réussissent parfois à cerner certains phénomènes biologiques, particulièrement ceux liés à l'évolution, de manière souvent plus fine que les heuristiques de la première catégorie, ou même que les algorithmes exacts. Nous avons en effet fait remarquer (section 4.1.1.1) que les alignements ayant un score optimal ne correspondent pas toujours aux alignements qui sont 'corrects' biologiquement. Le motif en est fondamentalement que certaines définitions de ressemblance données au chapitre précédent, bien qu'établies sur la base de critères mathématiques clairs et précis, représentent parfois une simplification des mécanismes biologiques.

4.1.3.2 Les heuristiques proches des algorithmes exacts

4.1.3.2.1 Les heuristiques sans garantie de la qualité de l'alignement obtenu

D'une certaine façon, nous avons déjà présenté des heuristiques au calcul du score d'un alignement optimal dans la mesure où les algorithmes exacts qui essayent de réduire le nombre de cellules à calculer dans le tableau de programmation dynamique utilisent parfois des bornes supérieures 'inexactes'. C'est le cas par exemple de Altschul *et al.* et de Gupta *et al.* pour les alignements ayant pour base des scores SP.

Les deux heuristiques que nous présentons ici cependant ne s'appuient pas sur le calcul de bornes supérieures ou inférieures. Ce ne sont donc plus des techniques de 'branch-and-bound'. Par ailleurs, ces heuristiques travaillent avec des scores obtenus à partir d'un arbre et non plus des scores SP. Elles ont pour principe l'adoption d'un arbre phylogénétique qui permette de simplifier les calculs. Une première approche est due à Sankoff et Cedergren [Sankoff and Cedergren, 1983] et utilise la formule de récurrence donnée à la page 86 du chapitre précédent. Celle-ci comporte une double minimisation (dans le cas d'un arbre, on travaille avec des mesures de distances) et l'idée de Sankoff et Cedergren est de limiter le nombre de calculs à effectuer pour obtenir la minimisation interne, c'est-à-dire limiter le nombre de comparaisons à effectuer entre chaînes de départ et chaînes ancestrales. Il faut pour cela choisir comme arbre phylogénétique un arbre décomposable en un ensemble de sous-arbres qui ont chacun trois branches et qui se superposent, puis appliquer l'algorithme exact uniquement à chacun de ces sous-arbres pris dans un certain ordre. En termes pratiques, ce qui change dans la façon de calculer le min interne de la formule est que, au lieu de réaliser, pour toute affectation possible de symboles des chaînes de départ (ou de trous) aux feuilles de l'arbre, toutes les attributions possibles de symboles ou trous aux nœuds internes, un de ceux-ci, notons-le x, est choisi initialement au hasard parmi ceux qui sont liés à deux autres nœuds auxquels des symboles a et b ont déjà été attribués. Un symbole est alors affecté à x suivant la valeur de a et b, puis un second nœud interne est choisi vérifiant les mêmes conditions que le nœud x et ainsi de suite. La qualité de l'alignement obtenu à la fin dépend bien sûr fortement de l'ordre avec lequel les nœuds internes sont sélectionnés (dans le cas où plusieurs choix sont possibles) pour se voir affecter un symbole ou un trou. La complexité de l'algorithme de Sankoff et Cedergren est $O((2n)^3.N)$.

Une approche différente est fournie par Waterman et Perlwitz [Waterman and Perlwitz, 1984] et permet d'aligner N chaînes de longueur moyenne n en $O(N.n^2)$. L'arbre phylogénétique utilisé est dans ce cas un arbre binaire et un concept nouveau appelé une 'chaîne moyenne pondérée' est employé (ces chaînes sont situées aux nœuds internes de l'arbre). Une chaîne moyenne pondérée de plusieurs chaînes définies sur un alphabet Σ est une chaîne $a = a_1a_2...a_k$ où chaque a_i a la forme $a_i = (p_0, p_1, p_2, ..., p_{|\Sigma|})$, avec p_0 correspondant à la proportion de trous dans les chaînes alignées résumées par la chaîne a et relatifs à la position i de a, et p_j pour $j > 0$ représentant la fréquence du $j^{ème}$ élément de Σ dans cet alignement, toujours relativement à la position i de a. Par conséquent, on a $p_j \geq 0$ et $\sum_{j>0} p_j = 1$. La ressemblance de deux 'lettres' $a = (p_0, p_1, ..., p_{|\Sigma|})$ et $b = (q_0, q_1, ..., q_{|\Sigma|})$ est donnée par :

$$d(a,b) = (\sum_{i \geq 0} w_i . \mid p_i - q_i \mid^{\alpha})^{\frac{1}{\alpha}}$$

où les w_i sont des facteurs de pondération et $\alpha \geq 1$ est une constante. Il a été montré que d représente en fait une métrique. La méthode commence donc par construire la 'chaîne moyenne' de deux des chaînes de départ placées aux feuilles de l'arbre binaire et qui sont les fils d'un même nœud de cet arbre. Puis le processus de construction se poursuit en suivant les relations de dépendances de l'arbre jusqu'à la racine pour laquelle une chaîne moyenne pondérée finale est dérivée. L'alignement des chaînes de départ est obtenu alors par un alignement de celles-ci avec la chaîne moyenne finale. La complexité en temps de l'algorithme de Waterman et Perlwitz est $O(N.n^2)$ et celle en espace $O(n^2 + n.N)$.

4.1.3.2.2 Les heuristiques garantissant une certaine qualité à l'alignement obtenu

Les deux heuristiques que nous présentons maintenant ont été élaborées par Gusfield [Gusfield, 1993]. Comme pour les heuristiques de la section précédente, celles-ci ont pour principe limiter le nombre de chaînes à comparer, et donc le nombre de calculs à effectuer. Elles différent des méthodes antérieures d'une manière importante cependant dans la mesure où elles sont capables également de fournir une borne supérieure à l'erreur qui est commise dans l'alignement obtenu, c'est-à-dire en fait dans le score de l'alignement produit par rapport à l'alignement optimal. Afin de pouvoir obtenir une telle borne, la mesure de ressemblance utilisée doit vérifier l'inégalité triangulaire. Pour faciliter les explications, nous considérons ici (dans les exemples en particulier) que cette mesure représente une simple distance d'édition bien que d'autres métriques ou ultramétriques puissent être utilisées (notons que la borne est indépendante de la mesure choisie).

Par rapport à un ensemble de départ comportant N chaînes, Gusfield garantit alors que :

$$score(\mathcal{A}) \leq (2 - \frac{2}{N}).score(\mathcal{O}) \leq 2.score(\mathcal{O})$$

où $\mathcal{O}$ est un alignement optimal des N chaînes et $\mathcal{A}$ l'alignement multiple réalisé par l'algorithme de Gusfield. Cette borne vaut pour les deux heuristiques qu'il présente bien que la première fasse appel à des scores SP et la seconde à des scores obtenus à partir d'un arbre.

La méthode s'appuyant sur des scores SP utilise le concept d'étoile centrale ('Center Star') que nous avons introduit dans le chapitre 3, section 3.4.1.5. Elle consiste à choisir à tour de rôle chacune des N chaînes de départ comme chaîne de référence, notons cette chaîne sc, et à considérer la somme

$$\sum_{\substack{i \in [1..N] \\ i \neq c}} distance(sc, si)$$

des distances de toutes les chaînes à sc. L'alignement des chaînes $s1, ..., sN$ fourni par l'algorithme est alors celui qui minimize cette somme et l'étoile centrale est une chaîne pour laquelle ce minimum est atteint (il peut y avoir plus d'une chaîne dans cette situation). Un exemple a été donné dans la figure 3.22.

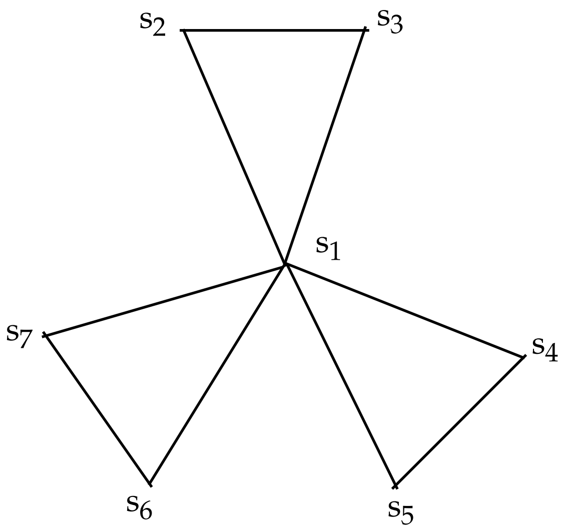


Figure 4.9: Un exemple de 3-étoile.

Comme nous l'avions fait observer alors, cette mesure de ressemblance peut être vue comme une approximation des scores SP ou comme une mesure intéressante et importante en soi. Gusfield lui-même la présente comme une approximation et c'est pourquoi nous avons placé son algorithme parmi les heuristiques. Nous argumenterons dans le chapitre 5 qu'une telle mesure de score multiple peut être plus valable dans certains cas, en particulier pour les comparaisons locales, que les scores SP.

Comme l'algorithme de Gusfield s'appuie sur un calcul de toutes les distances deux à deux, sa complexité en temps est en $O(n^2.N^2)$. Si uniquement $p < N$ chaînes sont choisies au hasard comme candidates possibles à devenir des étoiles centrales, l'erreur commise peut être plus grande mais Gusfield garantit néanmoins que l'on aura :

$$score(\mathcal{A}) \leq ((2 - \frac{2}{N}) + (\frac{N}{p} - \frac{1}{p})).score(\mathcal{O}).$$

Enfin, en travaillant non avec des alignements deux à deux mais avec des alignements L à L (c'est-à-dire, en construisant une L-étoile — voir figure 4.9 pour $L = 3$) Pevzner [Pevzner, 1992a] [Pevzner, 1992b] et Bafna [Bafna *et al.*, 1994] montrent que la borne pour l'erreur devient :

$$score(\mathcal{A}) \leq (2 - \frac{L}{N}).score(\mathcal{O}).$$

La complexité de l'algorithme est dans ce cas $O(N^3.n^L)$.

La seconde heuristique introduite par Gusfield cherche à construire un alignement des chaînes de départ en utilisant cette fois un arbre. Contrairement à ce qui se passe dans l'approche de Sankoff ou celle de Waterman, cet arbre ne comporte pas d'ancêtres. Les chaînes de l'ensemble se situent ainsi non seulement aux feuilles mais aussi aux nœuds internes de l'arbre. Aucun nœud interne correspond donc à une chaîne qui serait inconnue initialement. La méthode consiste alors à partir d'un graphe complet G à N nœuds dont chacun représente une chaîne de l'ensemble de départ et où chaque arête a pour poids le score d'un alignement optimal

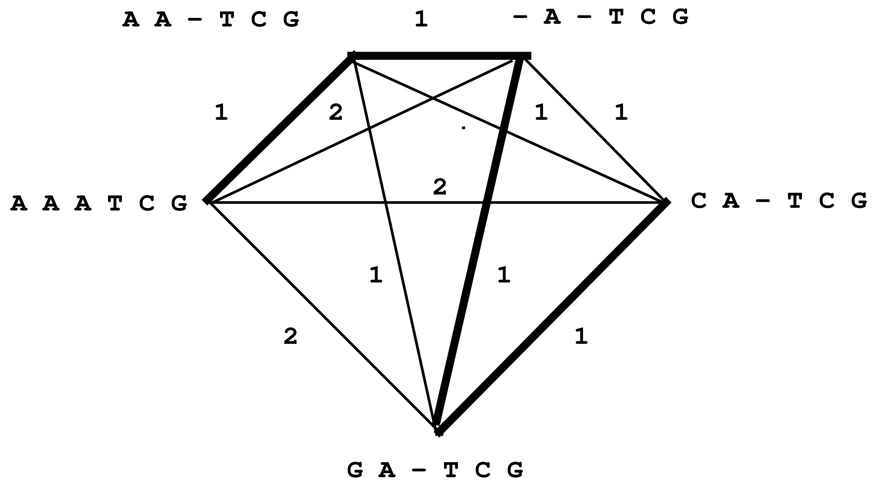


Figure 4.10: Un exemple de l'heuristique de Gusfield utilisant un arbre.

des deux chaînes liées par l'arête, et à chercher un arbre couvrant de score minimal de G (en anglais 'minimal spanning tree'). Il s'agit d'un arbre qui contient tous les nœuds de G et dont la somme des arêtes est la plus petite possible (voir la figure 4.10). Des algorithmes classiques en théorie des graphes (dus à Kruskal et Prim notamment [Kruskal, 1956] [Prim, 1957]) permettent de déterminer un tel arbre en un temps $O(N^2)$ (pire cas pour l'algorithme de Prim). L'alignement final des N chaînes est celui qui est consistant avec cet arbre. C'est donc celui dont le score de la projection sur chaque paire de chaînes (si, sj) est égal à la somme du poids des arêtes liant si et sj dans l'arbre. Comme toutes les distances entre paires de chaînes doivent être calculées, la complexité de l'algorithme est $O(n^2.N^2 + N^2)$, c'est-à-dire $O(n^2.N^2)$.

4.1.3.3 Les heuristiques par alignements progressifs

4.1.3.3.1 Introduction

Les heuristiques de la seconde catégorie (voir la section 4.1.3.1) possèdent un premier point en commun dans la mesure où toutes effectuent un alignement progressif des chaînes d'un ensemble de départ. Elles commencent donc par aligner une chaîne avec une autre, puis une chaîne avec un alignement obtenu au préalable, parfois un alignement avec un autre alignement. Le score d'un alignement de plus de deux chaînes ne représente donc que rarement la somme des scores de tous les alignements deux à deux induits. Ces heuristiques optimisent presque toujours les scores de tous les alignements deux-à-deux mais il n'y a plus d'optimisation globale de l'alignement multiple.

Ainsi que le signale Gusfield [Gusfield, 1995], ces méthodes progressives déterminent une suite de fusions de sous-ensembles disjoints des chaînes de départ qui peut être résumée par un arbre. Si les divers critères utilisés pour fusionner les sous-ensembles de chaînes sont supposés refléter l'histoire évolutive de celles-ci (par exemple, s'ils s'appuient sur l'hypothèse que des chaînes plus similaires entre elles sont également plus proches évolutivement), alors l'arbre obtenu représente non seulement une description du processus de construction de l'alignement final mais est aussi un arbre phylogénétique déduit des séquences fournies en entrée de l'algorithme. Les méthodes progressives d'alignement peuvent ainsi jouer un rôle important aussi bien dans la classification des macromolécules et l'identification de structures ou fonctions communes, que dans la déduction de phylogénies. Toutes ces approches sont donc bien sûr motivées par la recherche d'une plus grande efficacité algorithmique, mais certaines représentent aussi une réflexion parfois profonde sur la ressemblance de macromolécules et qui n'est pas moins valable du fait qu'elle est moins clairement définissable (une question ouverte — et que pose en tout cas Higgins [Thompson *et al.*, 1994a] — est alors de savoir si une telle réflexion pourrait être formalisée).

Nous pouvons répartir ces heuristiques par alignements progressifs en deux grandes classes qui sont :

1. celles qui effectuent de tels alignements en suivant un certain ordre, en général séquentiel, établi sur les chaînes de départ et qui alignent toujours deux chaînes ou une chaîne avec un alignement réalisé au préalable;

2. celles qui essayent de regrouper les chaînes suivant leur degré de ressemblance avant de les aligner (qui essayent en fait, implicitement ou explicitement, d'estimer un arbre phylogénétique pendant le déroulement de l'algorithme) et qui sont ainsi amenées à effectuer des alignements d'alignements.

Les secondes diffèrent des premières essentiellement parce que les arbres qu'elles construisent sont plus complexes. Il est important d'observer que toutes ces approches, particulièrement celles de la seconde classe, sont en fait des heuristiques à mettre en rapport avec les algorithmes exacts qui chercheraient à obtenir un alignement multiple sur la base d'un arbre qui ne serait pas connu à l'avance.

Notons qu'un second point en commun présenté par ces heuristiques vient du fait qu'elles ont besoin de pré-calculer les scores de tous les alignements deux à deux des chaînes de départ, ce qui explique que le terme dominant dans leur complexité soit le plus souvent en $n^2.N^2$. Les constantes peuvent bien sûr varier considérablement d'une méthode à l'autre, elles ne sont pas discutées ici.

4.1.3.3.2 Alignements progressifs suivant un ordre séquentiel

La façon sans doute la plus naturelle de procéder afin d'aligner N chaînes suivant un ordre séquentiel consiste à attribuer à chaque paire (si, sj) un score représentant une mesure de similarité ou de dissimilarité (ou distance) entre si et sj et à classer les paires d'après ces scores [Barton and Sternberg, 1987a] [Barton and Sternberg, 1987b] [Barton, 1990] [Barton and Sternberg, 1990] [Subbiah and Harrison, 1989]. Les deux chaînes les plus proches (les plus 'semblables') sont alors alignées en premier, notons-les $s1$ et $s2$, puis on aligne avec l'alignement de $s1$ et $s2$ la chaîne parmi celles qui demeurent qui est la plus proche soit de $s1$, soit de $s2$ et ainsi de suite. À la fin de chaque étape, lorsqu'un alignement de p parmi les N chaînes de départ a été obtenu, une série précise de raffinements successifs peut être appliquée à cet alignement.

Ainsi, si $3 \leq p \leq N$ chaînes ont été alignées dans un certain ordre ($s1$ avec $s2$, puis $s3$ avec l'alignement de $s1$ et $s2$, et ainsi de suite jusqu'à sp avec l'alignement de $s1, ..., s(p-1)$), cet alignement est 'gelé' en relation à tous les sous-ensembles possibles de q parmi les p chaînes pour q allant de 2 à $p-1$, et les $p-q$ chaînes restantes réalignées par rapport à l'alignement 'gelé' des q. Le coût excessif d'une telle opération (qui demande $O(N!)$) comparaisons deux à deux entre chaînes ou entre alignements) fait qu'en pratique quelques-uns seulement parmi tous les réalignements possibles sont effectués, ou bien alors le processus de raffinements est arrêté avant la fin. Subbiah opte pour la seconde approche puisqu'il fixe à l'avance le nombre maximal de fois où le processus peut être répété. Dans le cas de Barton par contre, la limitation est réalisée en fixant q égal à $p-1$, il n'y a donc jamais qu'une chaîne qui est 'libérée' et réalignée avec les $p-1$ autres.

Notons que dans les deux cas, l'ordre parmi les paires de chaînes peut être établi sur la base non de leurs scores de ressemblance mais plutôt de la signification statistique de ces scores où cette signification est calculée par rapport aux chaînes de l'ensemble, les symboles de chacune étant ré-ordonnés de manière aléatoire.

Enfin, Bains [Bains, 1986] [Bains, 1989] réalise des alignements progressifs suivant un ordre non complètement séquentiel. Dans son cas, une chaîne, appelons-la $s0$, est initialement privilégiée parmi toutes les autres et c'est l'ordre suivant lequel ces autres 'ressemblent plus' à $s0$ qui détermine laquelle va être sélectionnée pour être alignée avec $s0$, ou avec un alignement de $s0$ et des chaînes les plus proches de $s0$ qui auront été choisies au préalable. Une fois un premier alignement des N chaînes obtenu, une chaîne consensus est dérivée de cet alignement. Si elle est différente de la chaîne $s0$ de la première étape, elle prend sa place et tout le processus est répété jusqu'à convergence. Celle-ci peut bien sûr ne jamais se produire si, par exemple, les chaînes à aligner sont trop dissemblables, ou si le choix initial de $s0$ conduit à un processus qui boucle sur lui-même.

4.1.3.3.3 Alignements progressifs regroupant les chaînes ou estimant un arbre

Parmi les méthodes qui éventuellement regroupent certaines des chaînes de départ [Corpet, 1988] [Feng and Doolittle, 1987] [Feng and Doolittle, 1990] [Higgins and Sharp, 1988] [Higgins and Sharp,1989] [Higgins *et al.*, 1992] [Hogeweg and Hesper, 1984] [Smith and Smith, 1990] [Smith and Smith, 1992] [Taylor, 1987] [Taylor, 1990] [Thompson *et al.*, 1994a] [Thompson *et al.*, 1994b] [Wong *et al.*, 1993] la différence porte d'abord sur la façon d'effectuer ces regroupements. Comme pour les approches décrites dans la section précédente, initialement les scores de tous les alignements deux à deux des chaînes de l'ensemble sont calculés. Signalons que dans le cas de Higgins cependant, ce calcul est approché et non exact car il veut pouvoir traiter un très grand nombre de chaînes simultanément. Les chaînes peuvent alors être regroupées dès le début ou bien les groupes se former au fur et à mesure que l'alignement de l'ensemble se poursuit. Cela est le cas de Corpet en particulier qui commence par aligner les deux chaînes les plus proches, puis la matrice de tous les scores deux à deux est en partie recalculée avec la ligne correspondant à la chaîne $s2$ supprimée et celle correspondant à la chaîne $s1$ substituée par la moyenne des scores des lignes relatives à $s1$ et $s2$. Sont alignées ensuite les deux chaînes, ou la chaîne et un alignement préalable ou encore les deux alignements qui sont les plus proches d'après cette nouvelle matrice. Feng, Higgins, Hogeweg, Smith, Taylor et Wong de leur côté suivent la première approche, c'est-à-dire ils peuvent être amenés à effectuer les regroupements dès l'étape initiale et font en fait appel explicitement à des algorithmes de construction d'arbres phylogénétiques [Blanken *et al.*, 1982] [Klotz *et al.*, 1979] [Klotz and Blanken, 1981] [Saitou and Nei, 1987] [Sneath and Sokal, 1973] ou de regroupements hiérarchiques basés sur des graphes aléatoires [Wong *et al.*, 1993]. L'algorithme de Smith *et al.* produit en outre à la fin une chaîne consensuelle qui est toutefois différente de la chaîne moyenne de Waterman (voir la section 4.1.3.2.1). Dans leur cas, cette chaîne est définie non plus sur l'alphabet des monomères mais sur une couverture qui représente une partition de cet alphabet. La chaîne est ainsi un motif dégénéré (joker $\{\Sigma\}$ non autorisé) dans le sens de l'exemple de la figure 3.24 du chapitre 3.

Parmi tous ces algorithmes, celui de Higgins composant la série des CLUSTALs (appelé CLUSTAL W dans sa dernière version) est de loin le plus utilisé. La raison de ce succès est qu'il incorpore une richesse d'information dans son processus d'alignement qui n'est sans doute égalé par aucune autre méthode travaillant uniquement avec les séquences. Ainsi, bien que CLUSTAL W suive les mêmes principes que les méthodes antérieures (estimation d'un arbre, regroupement des chaînes selon cet arbre, alignement des chaînes intra puis inter-groupes), un soin très grand est accordé à tous les détails de la construction de l'arbre et au choix des valeurs à attribuer aux divers paramètres pouvant avoir une influence sur la qualité de l'alignement final, en particulier le poids à affecter aux trous. En ce qui concerne l'arbre par exemple, ce n'est pas seulement la position des chaînes les unes par rapport aux autres qu'Higgins *et al.* essayent de déterminer ainsi que le font les autres (c'est-à-dire la forme de l'arbre) mais aussi la position absolue de ces chaînes (recherche de l'emplacement de la racine de l'arbre) et le degré de leur dérive relative (affectation d'un poids aux branches de l'arbre suivant les distances des chaînes par rapport à l'ancêtre commun présumé placé à la racine). Ce dernier point permet à la fois de tenir compte de la présence éventuelle d'un biais dans l'ensemble à aligner (si plusieurs des chaînes sont très proches les unes des autres par exemple) et d'utiliser des matrices de substitutions appropriées suivant la distance séparant les chaînes. Le poids à attribuer aux trous quant à lui varie suivant que le trou vient d'être ouvert ou qu'il est simplement allongé, il dépend également de la distance séparant les chaînes (le poids est plus élevé pour les chaînes plus proches), de la longueur de ces chaînes (ce qui affecte le coût relatif à l'ouverture d'un trou), de la position du trou (le poids est plus élevé pour un trou introduit dans un alignement à un endroit où jusqu'alors il n'y en avait pas, il est plus faible dans les régions hydrophiles des protéines qui correspondent le plus souvent aux boucles etc) et enfin de l'environnement du trou (les monomères situés juste avant et après le trou). Comme le dit clairement Higgins, en particulier dans [Thompson *et al.*, 1994a], ces détails de construction de l'arbre et choix des valeurs à affecter aux paramètres ne sont guidés que par l'observation et l'expérience et demeurent donc de nature empirique pour l'instant.

4.1.3.3.4 Avantages et inconvénients

Indépendamment des informations supplémentaires qui peuvent être incorporées dans un alignement par ces méthodes d'alignements progressifs du fait qu'elles ne cherchent plus à optimiser une fonction particulière, une idée dominante qui est commune à toutes et sur laquelle nous voulons insister parce qu'elle est importante, est celle de comparer d'abord les objets qui se ressemblent le plus, et puis d'une certaine façon d''apprendre' quelque chose de cette première comparaison avant de se lancer à comparer les autres objets d'un ensemble. Cette idée est sans doute illustrée le mieux par la phrase de Feng [Feng and Doolittle, 1987] : "*once a gap — always a gap*" qui signifie qu'une fois qu'un trou a été introduit dans l'alignement de p des N chaînes de départ, celui-ci devrait être conservé dans l'alignement final si les p chaînes alignées en premier représentent celles qui sont les plus proches les unes des autres. Cette phrase de Feng qui ne s'applique qu'aux trous peut en fait être généralisée à d'autres facteurs jouant pour la qualité d'un alignement et ce qu'elle exprime de la façon la plus large est que la comparaison d'un ensemble d'objets devrait commencer par essayer de distinguer les parties communes qui sont les 'plus similaires'. Cette idée n'est pas incontestable. On peut argumenter en effet que l'identification des blocs fortement similaires (comportant moins d'erreurs par exemple) qui ne seraient présents que dans un certain pourcentage des objets que l'on veut comparer pourrait empêcher de repérer des blocs plus faiblement similaires (c'est-à-dire, pr'esentant plus d'erreurs) mais communs à pourcentage plus grand de ces objets. Cependant l'idée de comparer ce qui est le plus similaire d'abord est une idée puissante et s'applique aussi bien au cas global (comparer d'abord les chaînes qui se ressemblent le plus) qu'au cas local (repérer les mots communs correspondant à ce qui est le mieux conservé parmi un ensemble de ces chaînes). Dans le cas local cependant, localiser les similarités les plus fortes (ou les plus longues) d'abord n'empêche que temporairement de trouver les plus faibles (ou les plus courtes), et peut en fait être utilisée pour faciliter cette seconde recherche.

Pour en finir avec l'approche globale et progressive, observons que, outre son manque de formalisme, celle-ci présente trois principaux inconvénients qui sont :

1. le fait que l'alignement final dépend de l'ordre dans lequel les alignements intermédiaires sont effectués — ceci est particulièrement vrai pour les approches séquentielles mais s'applique aussi aux méthodes qui établissent un arbre phylogénétique;

2. bien qu'aucune de ces approches ne cherche vraiment à optimiser un score ou un alignement, toutes peuvent conduire à des situations de 'blocage' où l'algorithme produit un alignement de médiocre qualité et ne réussit plus à progresser vers un meilleur alignement;

3. la qualité des alignements réalisés dépend fortement du choix effectué sur certains paramètres (comme les scores de substitution, les poids à affecter aux trous etc), la valeur à attribuer à ces paramètres étant particulièrement difficile à évaluer lorsqu'il s'agit non plus d'aligner deux chaînes, mais d'aligner une chaîne avec un alignement antérieur, ou d'aligner 2 alignements précédemment obtenus.

Observons que ce dernier problème concernant la difficulté de choisir les poids à affecter à divers paramètres, bien qu'il soit plus aigu ici, est vrai aussi dans le cas des algorithmes exacts (voir le chapitre 3). Nous pouvons ainsi définir de manière mathématiquement très précise ce que signifie l'alignement optimal de plusieurs chaînes, mais la définition d'une fonction de score multiple repose en dernière instance sur le choix des scores deux à deux. Un changement dans ces scores change l'alignement, même si celui-ci demeure optimal par rapport au choix qui a été fait. Plusieurs des définitions de ressemblance que nous avons données au chapitre 3 ont pour base ces scores, qui ne s'appuient en fin de compte sur aucune notion qui paraisse biologiquement tout à fait fiable.

4.2 Approche locale

4.2.1 Introduction

Tout ce que nous avons présenté dans la section 4.1 concernait les méthodes directes de comparaison globale de chaînes. Or notre intérêt porte surtout sur les comparaisons locales. Les approches que nous venons de décrire sont importantes cependant car en fait la plupart du temps les méthodes locales ne sont que des adaptations de celles utilisées pour comparer des chaînes dans leur totalité. D'une manière très grossière, nous pouvons dire que la différence porte souvent simplement sur le moment où s'arrête une comparaison. Le principe de base est ainsi le même pour un grand nombre d'approches locales bien que les variations puissent être considérables.

Lorsque nous avons présenté les méthodes globales, nous avons aussi commencé par montrer les algorithmes exacts et ce n'est qu'ensuite que nous avons parlé des heuristiques. La raison en est qu'historiquement c'est à peu près ainsi que les choses ont été développées. Les chercheurs ont d'abord été très 'ambitieux', ensuite ils ont du revoir leur ambition à la baisse. Les méthodes locales que nous allons décrire maintenant sont venues après les globales, ce n'est en effet que plus tard que l'on s'est rendu compte que la recherche des parties communes, et de ces parties seulement, pouvait être à la fois intéressante en soi et souvent aussi plus appropriée que la recherche d'une ressemblance globale. Or lorsque l'on en est arrivé là quelques leçons avaient déjà été apprises et ce sont ainsi les heuristiques qui ont été élaborées en premier. C'est donc celles-ci que nous présentons d'abord.

Avant cela, il convient d'observer que même lorsque les chercheurs sont d'accord pour dire que, dans certaines situations, ce sont les ressemblances locales qui sont les plus, ou parfois les seules, importantes à identifier, tous ne sont pas d'accord sur la meilleure façon de procéder pour repérer ces ressemblances. Ainsi Posfai [Posfai *et al.*, 1989] en particulier considère que la meilleure manière de localiser des traits communs à un ensemble de chaînes est de les comparer globalement d'abord, puis d'extraire de ces comparaisons (i.e. alignements) les 'morceaux' les plus caractéristiques. Cette opinion est fortement contestée par d'autres ainsi que le signale Gusfield [Gusfield, 1995] mais il est important de la mentionner. La méthode de Posfai est également celle employée par Barton [Barton, 1990] [Barton and Sternberg, 1990].

Comme pour l'approche globale, les heuristiques qui recherchent les similarités locales à un ensemble de chaînes peuvent être de divers types. Parmi ceux-ci, on distingue deux grandes classes : les heuristiques qui cherchent à optimiser une fonction de score et qui utilisent des formes modifiées de programmation dynamique pour le faire, et les méthodes d'optimisation stochastique. En ce qui concerne la première approche, nous allons devoir revoir un peu les techniques de programmation dynamique mais appliquées au cas local cette fois. Nous allons donc présenter les méthodes exactes d'abord mais celles-ci ne sont jamais utilisées en pratique et nous n'en parlons que parce que les méthodes approchées s'en servent.

4.2.2 Les heuristiques

4.2.2.1 Méthodes d'optimisation d'une fonction de score

4.2.2.1.1 Une parenthèse sur les méthodes exactes : la programmation dynamique appliquée au cas local

Dans la section 4.1.1.1, nous avons vu comment, étant donné une fonction de score, rechercher le score d'un alignement optimal global de deux chaînes $s1 = a_1...a_n$ et $s2 = b_1...b_m$. Si nous voulons déterminer maintenant le score d'un alignement optimal local des chaînes $s1$ et $s2$ au sens de la définition 3.4.7 présentée dans le chapitre 3, il faut commencer par rappeler que nous devons supposer que les scores avec lesquels nous travaillons mesurent une similarité entre monomères. De plus, il est impératif qu'au moins un de ces scores soit négatif. Le score optimal local est alors le plus élevé observé parmi tous les alignements possibles de toutes les paires de mots possibles dans les deux chaînes [Smith and Waterman, 1981]. La formule d'induction sur les valeurs du tableau est dans ce cas donnée par :

$$\mathcal{D}(i,j) = max\{0, \mathcal{D}(i-1,j-1) + score\begin{pmatrix} a_i \\ b_j \end{pmatrix}, \mathcal{D}(i-1,j) + w_\lambda, \mathcal{D}(i,j-1) + w_\lambda\}$$
$$\mathcal{D}(i,0) = 0$$
$$\mathcal{D}(0,j) = 0$$

pour $1 \leq i \leq n$ et $1 \leq j \leq m$. Le score recherché est alors égal à $max_{i,j}\mathcal{D}(i,j)$. Par rapport à la formule dans le cas global, le changement porte sur les valeurs frontières du tableau, ainsi que sur le fait que $D(i,j)$ est réinitialisé à zéro lorsque les trois autres termes de la formule sont négatifs. Une paire de mots (il peut y en avoir plus d'une) pour laquelle ce maximum est atteint est retrouvée en suivant le tableau des pointeurs qui aura été construit en même temps que le tableau des scores, depuis la cellule correspondant au score maximal jusqu'à la première cellule rencontrée ayant un score nul (voir la figure 4.11).

recherche meilleur alignement local

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

modification de la matrice
(élimination du meilleur)
recherche du second meilleur

	-	G	C	T	C	T	G	C	G	A	A	T	A
-	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	2	0	2	0	0	2	0	0	0	0	0
G	0	2	0	1	0	1	2	0	4	2	0	0	0
T	0	0	1	2	0	2	0	1	2	3	1	2	0
T	0	0	0	3	1	0	1	0	0	1	2	3	1
G	0	2	0	1	2	0	0	0	2	0	0	1	2
A	0	0	1	0	0	1	0	0	0	4	2	0	3
G	0	2	0	0	0	0	3	1	0	2	3	1	1
A	0	0	1	0	0	0	1	2	0	2	1	2	3
T	0	0	0	3	1	2	0	0	1	0	1	0	1
A	0	0	0	1	2	0	1	0	0	3	2	0	0
C	0	0	2	0	3	1	0	3	1	1	2	1	0
T	0	0	0	4	2	5	3	1	2	0	0	4	2

Figure 4.11: Programmation dynamique appliquée au cas local - exemple extrait du livre de M. Waterman [Waterman, 1995] ($score(a_i, b_j) = 2$ si $a_i = b_j$, -1 sinon, w_λ = -$2l$ où l est la longueur du trou).

Il n'est pas tout à fait intuitif de comprendre pourquoi la formule de récurrence ci-dessus produit effectivement le résultat souhaité (le score d'une paire de mots la plus similaire parmi toutes les paires possibles, ainsi qu'une paire elle-même) aussi nous en fournissons une preuve.

Preuve que l'algorithme de Smith et Waterman est correct Prouvons d'abord que $max_{i,j}\mathcal{D}(i,j)$ est bien le score recherché.

Il suffit pour cela de prouver que $\mathcal{D}(i,j)$ représente pour $1 \leq i \leq n$ et $1 \leq j \leq m$ le meilleur score pour l'alignement optimal de toute paire de mots de $s1$ et $s2$ se terminant en i et j respectivement, c'est-à-dire, il suffit de prouver que :

$$\mathcal{D}(i,j) = max\ \{score(a_k...a_i, b_l...b_j) \mid 1 \leq k \leq i,\ 1 \leq l \leq j\}$$

où $score(a_k...a_i, b_l...b_j)$ est le score d'un alignement optimal global de $a_k...a_i$ et $b_l...b_j$.

Supposons par absurde que cela ne soit pas le cas. Il existe donc au moins une paire de mots $a_k...a_i$ et $b_l...b_j$ se terminant en i et j dans $s1$ et $s2$ respectivement avec $1 \leq k \leq i$ et $1 \leq l \leq j$ telle que $score(a_k...a_i, b_l...b_j)$ est strictement supérieur à $\mathcal{D}(i,j)$. Soit $\mathcal{D}'$ le tableau des scores/pointeurs de l'alignement global de $a_k...a_i$ et $b_l...b_j$. Quel que soit un chemin optimal dans $\mathcal{D}'$, ce chemin passe nécessairement par une cellule, notons-la (p,q) avec $k \leq p \leq i$ et $l \leq q \leq j$, telle que $score(a_k...a_p, b_l...b_q) < 0$ car si cela n'était pas le cas, nous aurions $\mathcal{D}'(i-k, j-l) = \mathcal{D}(i,j)$ ce qui contredirait l'hypothèse que $\mathcal{D}'(i-k, j-l) = score(a_k...a_i, b_l...b_j) > \mathcal{D}(i,j)$. Mais alors $score(a_{p+1}...a_i, b_{q+1}...b_j)$ serait strictement supérieur au score de $a_k...a_i$ et $b_l...b_j$ puisque $score(a_k...a_i, b_l...b_j) = score(a_k...a_p, b_l...b_q) + score(a_{p+1}...a_i, b_{q+1}...b_j)$ ce qui à son tour contredirait l'hypothèse que $(a_k...a_i, b_l...b_j)$ est une paire de mots se terminant en i et j dans $s1$ et $s2$ respectivement ayant une valeur maximale dans le tableau $\mathcal{D}$.

Il faut maintenant prouver que la procédure permettant de retrouver une paire de mots pour laquelle ce maximum est atteint est correcte, c'est-à-dire qu'elle fournit bien une paire de mots ayant un score maximal parmi toutes les paires de mots possibles de $s1$ et $s2$. La preuve de cela est immédiate. En effet, soit $(a_k...a_i, b_l...b_j)$ une paire de mots retrouvée par la procédure. Il est clair que $score(a_k...a_i, b_l...b_j) = \mathcal{D}(i,j)$ puisque la liste des pointeurs est parcourue jusqu'à ce que soit rencontrée la première cellule du tableau contenant un zéro et que, par définition de la récurrence, le score de toute paire de mots se terminant en $k-1$ et $l-1$ dans $s1$ et $s2$ respectivement est négatif. Comme $\mathcal{D}(i,j)$ est le score optimal local maximal, alors $(a_k...a_i, b_l...b_j)$ est une paire de mots recherchée.

$\square$

Les diverses économies qui peuvent être réalisées dans le calcul du score d'un alignement optimal de deux chaînes ne s'appliquent pas au cas des alignements locaux. Toutes les cellules de la matrice doivent donc être calculées, et l'algorithme de programmation dynamique est dans tous les cas en $O(n.m)$.

La détermination non pas uniquement du score d'un meilleur alignement, mais celle d'un alignement optimal lui-même possède la même complexité et est, comme nous venons de le voir, réalisée à l'aide d'un tableau de pointeurs construit en même temps que le tableau des scores. L'obtention par contre de tous les alignements optimaux locaux peut impliquer un recalcul des valeurs de la matrice (voir la figure 4.11 [Waterman and Eggert, 1987]). Il en est de même lorsque sont recherchés les c meilleures similarités locales, et dans ce cas la complexité est en $O(n.m.c)$ [Blum, 1990]. Toutefois ce recalcul n'est pas nécessaire lorsque seuls les c meilleurs scores nous intéressent (pas les alignements) et que le poids attribué à un trou est une fonction linéaire de sa longueur [Barton, 1993].

Enfin, en ce qui concerne la complexité en espace, elle est toujours en $O(n.m)$ lorsque les alignements sont recherchés, et en $O(min(m, n))$ lorsque l'on désire connaître uniquement le score optimal local [Barton, 1993] et que, à nouveau, le poids attribué à un trou est une fonction linéaire de sa longueur.

De même que le multiple global, le multiple local peut être vu comme une extension pure et simple de la programmation dynamique à N dimensions. *A priori*, rien n'empêche de considérer que les scores utilisés dans ce cas puissent être des scores calculés sur la base d'un arbre phylogénétique aussi bien que des scores SP. Les heuristiques qui s'appuient sur la programmation dynamique pour faire du multiple local utilisent cependant toujours des scores SP, et à notre connaissance aucune tentative n'a été faite à ce jour pour combiner le multiple local avec des scores ayant pour base un arbre connu ou estimé.

À nouveau ici, aucune des économies qu'il est possible de réaliser dans le calcul d'un score ou d'un alignement optimal global ne s'applique au cas local. La complexité en temps de l'agorithme est donc en $O(n^N)$ lorsque seul le meilleur score ou un meilleur alignement est recherché. La complexité en espace est quant à elle en $O(n^N)$ pour la recherche d'un meilleur alignement et en $O(n^{N-1})$ pour celui du meilleur score.

Ainsi que nous l'avons dit, une telle approche exhaustive n'est pas utilisée en pratique et n'a jamais en fait été formalisée comme dans le cas du deux à deux. Par contre, plusieurs heuristiques ont pour base de départ les principes de calcul de la programmation dynamique et ce sont elles que nous montrons dans la section suivante.

4.2.2.1.2 Les méthodes approchées

4.2.2.1.2.1 Les méthodes approchées pour du deux à deux — le cas particulier du criblage de banque

Les méthodes de comparaison locale dont nous allons parler d'abord font en réalité du deux à deux, dans un seul cas il y a comparaison de trois objets simultanément. Nous les plaçons malgré cela à cet endroit du chapitre car les algorithmes concernés ont à effectuer un très grand nombre de ces comparaisons deux à deux, et doivent ainsi faire face à des problèmes de temps d'exécution d'une amplitude similaire à ceux rencontrés par les algorithmes de comparaison multiple d'un ensemble de chaînes.

En effet, le sujet qui va nous concerner dans cette section peut être vu comme un cas un peu particulier d'une exploration de la ressemblance entre séquences macromoléculaires telle que présentée jusqu'ici. Désormais, parmi l'ensemble des chaînes qui doivent être traitées, une va se trouver privilégiée par rapport aux autres. Cette chaîne privilégiée (qui est appelée chaîne requête — en anglais 'query sequence') représente le plus souvent une macromolécule dont la séquence aura été récemment déterminée mais dont on ignore encore quelle fonction elle exerce et à quelle famille elle appartient. Afin d'essayer d'établir son 'identité' et le rôle qu'elle joue, elle est donc comparée à d'autres séquences pour lesquelles ces informations sont connues et qui se trouvent placées dans une banque. Or les banques actuelles contiennent une quantité considérable d'éléments (52 205 séquences pour les banques de protéines (SWISS PROT, version 33, Février 1996) et 701 246 pour les banques d'acides nucléiques (EMBL, version 46, Mars 1996), représentant 18 531 384 acides aminés et environ 473 000 000 bases nucléiques respectivement).

Il était jusqu'à récemment hors de question d'utiliser un algorithme de programmation dynamique (le plus sensible pour des comparaisons deux à deux) afin de réaliser toutes les comparaisons possibles entre la chaîne privilégiée et les chaînes de la banque. La complexité d'une telle analyse, appelée un criblage de banque, est en $O(n.m.M)$ où n est la longueur de la chaîne requête, m est la longueur moyenne des chaînes de la banque et M est le nombre de celles-ci (observons que $M >> N$ où N est le nombre moyen de chaînes traitées par les algorithmes de comparaison multiple). Bien sûr, l'introduction d'implémentations parallèles de l'algorithme de programmation dynamique [Brutlag *et al.*, 1993] [Coulson *et al.*, 1987] a rendu ce problème moins aigu mais les techniques utilisées par les deux programmes de criblage de banque décrits ci-dessous présentent néanmoins un intérêt parce que les heuristiques qu'ils emploient ont souvent inspiré les algorithmes de comparaison locale de plusieurs chaînes que nous verrons plus loin, ou parce qu'ils utilisent une notion de ressemblance qui est intéressante et adaptable au cas multiple (définition 3.5.14). Tous deux peuvent être vus comme réalisant une approximation de la programmation dynamique à deux dimensions. La première méthode due à Pearson se ramène en fait à un simple comptage de monomères ou de k-mères identiques (qui peut être étendu au comptage de k-mères similaires par une relation R) tandis que la seconde (BLAST) effectue de la programmation dynamique sans trou à partir d'un certain nombre de 'noyaux de ressemblance' repérés au préalable par une technique basée sur la génération de 'synonymes'.

Nous commençons par décrire la plus élémentaire des deux méthodes, celle de Pearson.

Ce cribleur de banque élémentaire et relativement encore assez utilisé par la communauté biologique s'appelle FastN ou FastP suivant que les séquences traitées sont des séquences nucléiques ou des protéines, ou FastA dans sa dernière version pour protéines. Ce programme constitué ainsi par un ensemble de sous-programmes a été développé par Pearson et Lipman entre 1985 et 1988 [Lipman and Pearson, 1985] [Pearson and Lipman, 1988] [Pearson, 1990] [Pearson, 1991].

FastN/FastP et FastA reposent sur le même principe de fonctionnement. L'algorithme présente donc toujours une première phase de pré-traitement puis se divise en trois étapes. La première et la seconde sont communes aux deux programmes tandis que la troisième étape est propre à FastA. Nous ne décrivons ici que les deux premières qui forment l'essentiel de la méthode et esquissons seulement l'extension réalisée par FastA.

La phase de pré-traitement réalise la construction d'une simple table indexée des mots de longueur k (appelés des 'k-uples') de la chaîne requête, où k est une valeur fixée à l'avance (à 1 ou plus souvent 2 pour les protéines et à 4 ou 6 pour l'ADN/ARN). La taille de cette table est donc égale au nombre total de k-uples possibles, soit 4^k ou 20^k. À chaque k-uple est associé une valeur unique, c'est-à-dire un code identifiant l'entrée correspondante de la table et chaque entrée de la table contient la liste des positions sur la chaîne requête où ce k-uple particulier est rencontré.

La première étape de l'algorithme utilise alors une variante de la méthode des diagonales conçue par Wilbur et Lipman [Wilbur and Lipman, 1983] afin d'identifier ce qui est appelé les régions de similarité entre la chaîne requête r et chacune des chaînes b de la banque. Chacune de ces régions correspond à un glissement de r par rapport à b (voir la figure 4.12), puis à un alignement local, sans trous, des deux chaînes commençant et se terminant par un mot identique de taille supérieure ou égale à k. À l'intérieur des régions alignées peuvent alterner des mots identiques et d'autres qui ne le sont pas. Le nombre de ces derniers ne peut néanmoins dépasser une certaine valeur l fixée à l'avance (voir la figure 4.13). Parmi toutes ces régions de similarité locale, les dix ayant la plus grande densité, c'est-à-dire le plus grand nombre de k-uples identiques avec la chaîne requête, sont conservées (dans le cas où $k > 1$, les k-uples peuvent se chevaucher). Ceci revient à choisir les dix meilleurs alignements locaux, ou glissements. Tous les alignements sans trous possibles de la chaîne requête r avec une des chaînes b de la banque peuvent être visualisés à travers une matrice dont les colonnes représentent les symboles de

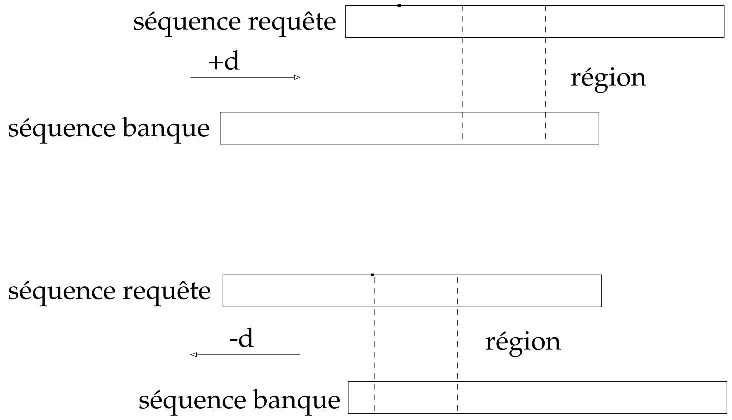


Figure 4.12: Glissement dans FastP/N et FastA.

la chaîne banque et les lignes celles de la requête. Cette matrice que nous dessinons dans la figure 4.13 n'est bien sûr pas effectivement construite. Dans l'algorithme, seul est utilisé un tableau des diagonales de la matrice (d'où le nom de la méthode) dont chacune correspond à un glissement de r par rapport à b. L'entrée d du tableau indique ainsi le nombre de k-uples (éventuellement chevauchants) appartenant à la dernière région de l'alignement obtenu par un décalage de d positions d'une chaîne relativement à l'autre. Par ailleurs, une file de priorité sert à conserver en permanence les informations permettant de repérer les dix meilleures régions (celles contenant le plus grande nombre de k-uples), à savoir leurs positions de début et fin sur chacune des chaînes.

La seconde étape consiste alors à utiliser une matrice de substitution, telle que PAM250, pour attribuer un score aux dix régions sélectionnées à la première étape. Ainsi le choix des régions à conserver est effectué sur la base d'un comptage de k-uples identiques (éventuellement séparés entre eux par des k-uples différents ainsi que nous l'avons vu) mais leur 'valeur' finale dépend de leurs scores dans le sens classique employé en programmation dynamique (sans trou - voir la definition 3.2.17). Observons que ce score porte en outre sur les k-uples identiques et sur les mots compris entre ces derniers dont le nombre est inférieur à l (c'est-à-dire, il porte sur toute une région). Un tri des régions d'après leur nombre de k-uples identiques peut ainsi ne pas être le même que celui obtenu d'après ces scores. Dans le cas de FastN/FastP, la ressemblance d'une chaîne de la banque par rapport à la chaîne requête est alors mesurée par le meilleur de ces scores.

La troisième étape de FastA consiste simplement à vérifier s'il n'est pas possible de joindre quelques-unes parmi les dix régions initialement identifiées afin de former un seul alignement ayant un score supérieur à celui des dix pris séparément. FastA essaye ainsi de réaliser une sorte d'alignement avec trous — les trous correspondant cependant aux segments de jonction entre deux régions, et seulement à ces segments. Les régions initiales sont préservées telles quelles, aucun trou n'étant permis à l'intérieur. Pour effectuer cet alignement, FastA emploie une technique de programmation dynamique [Pearson, 1990] et c'est le score de cette région qui, dans le cas de FastA, mesure la ressemblance de chacune des chaînes de la banque avec la chaîne requête.

Le résultat de l'algorithme quelle que soit la version consiste alors en la présentation d'une

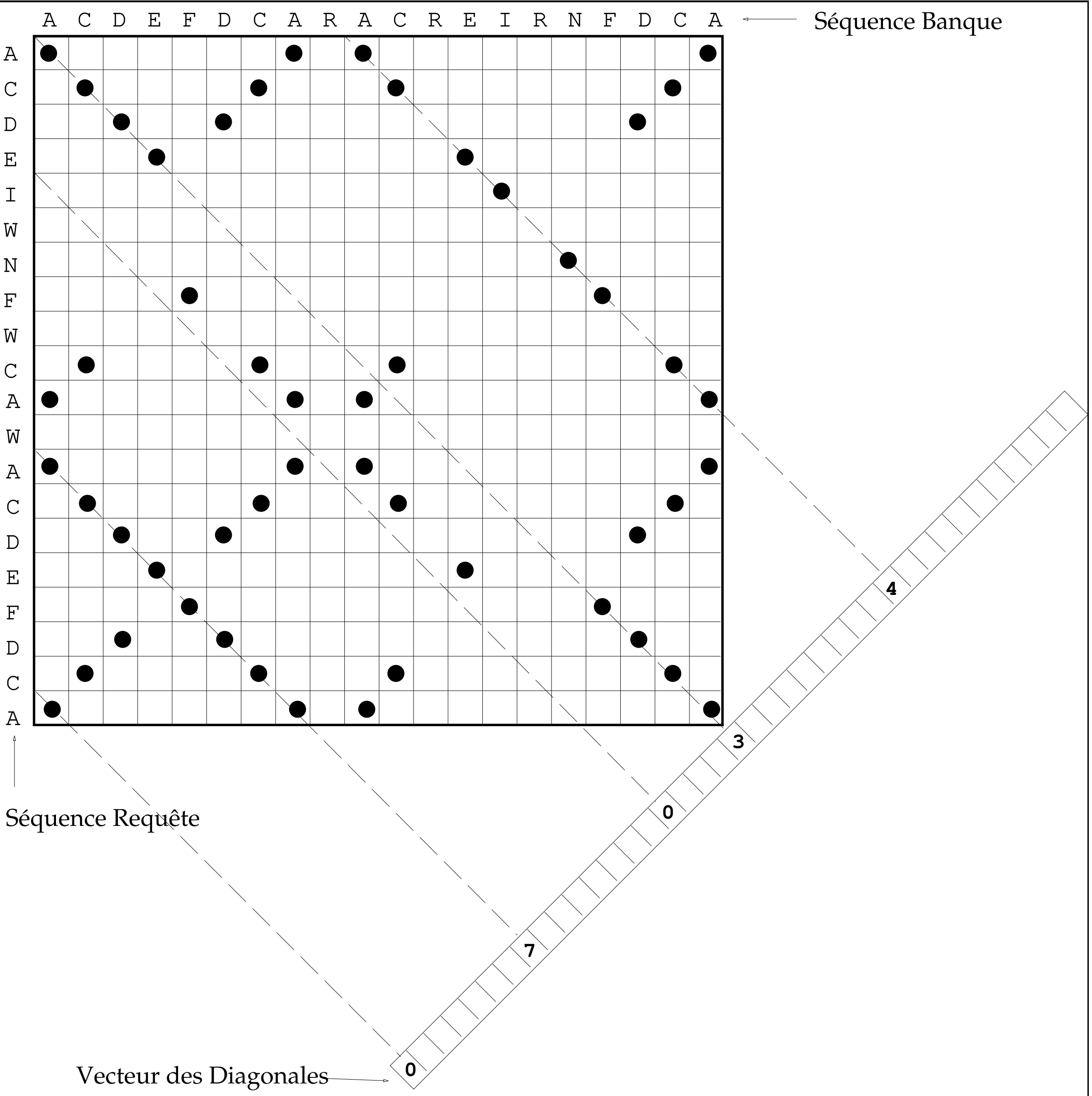


Figure 4.13: Matrice de FastN/FastP et FastA, jamais effectivement construite (ici, $k = 2$ et $l = 2$).

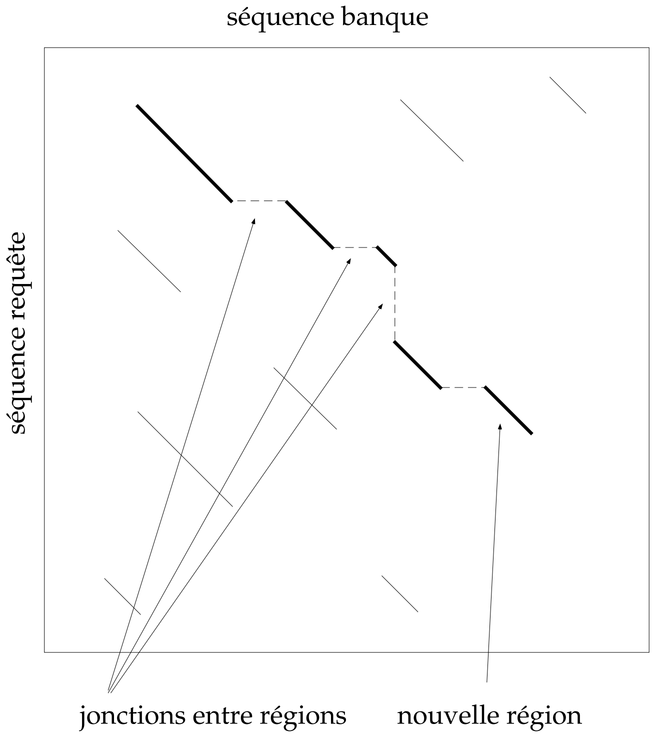


Figure 4.14: Regroupement des régions dans FastA.

```
/* Construction de la table indexée */
/* Entrée */
        n = longueur de la chaîne requête
        k = 2 dans cet exemple
/* Structure de données */
        V = tableau de taille | Σ |^k de pointeurs sur une liste de positions de la chaîne
        requête (V[i] pointe sur la liste de toutes les occurrences des mots de longueur k
        de la chaîne requête en relation de similarité avec l'élément de Σ^k codé par l'entier
        i)

        pour chaque facteur r_i r_{i+1} situé à la position i de la chaîne requête
                pour (a ∈ Σ)
                        si (a R r_i)
                                pour (b ∈ Σ)
                                        si (b R r_{i+1})
                                                placer la position i dans la liste V[code(ab)];
```

Figure 4.15: Phase de pré-traitement de FastN/FastP et FastA modifiés par Viari : Construction de la table indexée pour une relation de similarité R^k.

liste triée par les scores en ordre décroissant de toutes les chaînes de la banque pour lesquelles ce score est supérieur à un certain seuil fixé par l'utilisateur.

Comme nous pouvons le constater, ces programmes sont très simples mais relativement efficaces pour repérer des ressemblances lorsque celles-ci ne sont pas trop faibles. Il est également aisé d'introduire plus de souplesse dans la recherche en travaillant non plus avec une relation d'identité entre k-uples mais avec une relation de similarité.

Une première extension de l'algorithme a ainsi été réalisée par Brutlag [Brutlag *et al.*, 1990] pour laquelle deux k-uples $u_1 ... u_k$ et $v_1 ... v_k$ sont similaires si :

$$\mathcal{M}(u_1, v_1) \times \mathcal{M}(u_2, v_2) \times ... \times \mathcal{M}(u_k, v_k) \geq t^k$$

où $\mathcal{M}$ est une matrice de similarité et t est une valeur seuil.

Il est possible également d'établir une relation de similarité entre k-uples à partir d'une relation de similarité entre monomères comme celle décrite dans la section 3.4.3.2 du chapitre 3 [Viari and Pothier, 1993]. Dans ce cas, les k-uples $u_1, ..., u_k$ et $v_1, ..., v_k$ sont dits similaires si $u_i \; R \; v_i$ pour tout $i \in \{1, ..., k\}$.

En termes de l'algorithme, seule change la phase de pré-traitement, à savoir la construction de la table indexée à partir de la chaîne requête r. Celle-ci est évidemment plus remplie et nécessite plus de temps pour être parcourue. Une idée de l'algorithme pour cette phase est donnée dans la figure 4.15. La recherche des k-uples similaires entre r et chacune des chaînes de la banque puis la sélection des meilleures régions se fait de la même façon que pour FastN/FastP et FastA.

Voyons maintenant quelle est la complexité du programme de Pearson. La construction de la table indexée est au pire en $O(n. \mid \Sigma \mid^k)$ où n est la longueur de la chaîne requête. La recherche des k-uples quant à elle se fait en un temps $O(\frac{n.m.M.g^k}{|\Sigma|^k})$ où m est la longueur moyenne des chaînes de la banque et M est leur nombre, et avec bien sûr $g = 1$ pour les versions originales de FastN/FastP et FastA.

Nous passons maintenant au second programme de criblage de banque, BLAST, de loin le plus connu et utilisé. BLAST ('Basic Local Alignment Search Tool') a été développé par Altschul et ses collègues [Altschul *et al.*, 1990] comme une alternative à FastN/FastP et FastA. La philosophie suivie par ce programme est celle qui considère la valeur optimale d'une fonction de score comme étant la meilleure mesure de la ressemblance entre deux chaînes. Idéalement il faudrait donc pouvoir repérer ce que les auteurs appellent les 'paires de segments maximales' (c'est-à-dire de mots) entre la chaîne requête et chacune des chaînes de la banque. Une paire de segments est dite maximale (abrégé en MSP — en anglais 'Maximal Segment Pair'), si elle possède le plus grand score parmi toutes les paires de segments de longueur identique appartenant aux deux chaînes, où le score d'une paire est calculé à l'aide d'une matrice de similarité (voir la section 3.2.2.2). Les MSP correspondent donc tout simplement à ce que localise l'algorithme de programmation dynamique appliqué au cas local de Smith et Waterman (sans trou) [Smith and Waterman, 1981] (voir la section 4.2.2.1.1).

Bien sûr, afin de trouver les MSP de deux chaînes de manière exacte, il faudrait faire appel à cet algorithme dont la complexité en temps est proportionnelle au produit des longueurs des chaînes. Afin d'aller plus vite, il est alors nécessaire d'employer une heuristique. Celle de BLAST repose sur une observation très simple. Lorsque l'on effectue une recherche dans une banque contenant des milliers de chaînes, il arrive fréquemment que seul un petit nombre de ces chaînes, parfois même aucune, présente une similarité significative avec la chaîne requête. BLAST accélère donc la recherche dans une banque en minimisant le temps passé à analyser des chaînes dont le score de la MSP qui serait obtenue avec la chaîne requête a peu de chances d'excéder une certaine valeur S dont nous verrons plus loin comment elle est établie. Pour ce faire, il va considérer que les chaînes qui peuvent posséder une MSP avec un score supérieur à S sont uniquement celles qui contiennent au moins un mot de longueur w ayant un score d'au moins $T < S$ lorsque aligné, sans trous, avec un mot de la requête.

BLAST effectue ainsi initialement un simple filtrage de la banque qui a pour but d'éliminer toutes les chaînes ne contenant aucun mot de longueur w ayant un score d'au moins T avec un mot de la chaîne requête. Pour cela, il commence par considérer tous les mots de longueur w de la chaîne requête et établit une liste qui, pour chacun d'eux, indique les éléments de Σ^w dont le score avec le mot de la requête excède T. Les mots qui sont placés dans cette liste sont appelés des w-mères. La liste finale peut représenter un volume important (typiquement plus de 10000 mots) mais elle peut être produite en un temps directement proportionnel au nombre d'éléments qu'elle contient. Observons également qu'un mot de la chaîne requête peut être représenté par plusieurs mots dans la liste, ou, au contraire, par aucun. En effet, rappelons que dans certaines matrices de similarité (par exemple la matrice PAM250 donnée dans la figure 3.3) la substitution d'un acide aminé par lui-même ne possède pas toujours le score le plus élevé. Aussi, suivant la valeur fixée pour T, certains mots de la chaîne requête peuvent avoir un score en-dessous de T lorsqu'ils sont alignés avec eux-mêmes. Enfin la liste peut être préservée sous forme d'une table indexée ou sous forme d'un automate minimal. Quel que soit son mode de représentation, une fois cette liste construite, une recherche séquentielle de chacune des chaînes de la banque permet de décider rapidement si celle-ci contient un mot de longueur w qui forme une paire avec un mot de la chaîne requête ayant un score supérieur ou égal à T. Si cela est le cas, l'adresse de ces mots (appelés des 'hits') est conservée pour la deuxième phase de l'algorithme.

Dans cette seconde phase, BLAST essaye de prolonger (à gauche et à droite) tout mot ('hit') ainsi trouvé sur une des chaînes de la banque afin de déterminer si la paire de score supérieur à

T qu'il forme avec un des mots de la chaîne requête n'est pas contenue dans une paire de mots plus longs possédant un score supérieur ou égal à S. Le prolongement des mots initiaux se fait également sans trous. Il est clair que plus petite est la valeur choisie pour T, plus grande est la probabilité qu'une paire de mots ayant un score d'au moins S contienne une paire de mots ayant un score d'au moins T et ne soit donc pas 'ratée'. Cependant, une petite valeur pour T a aussi pour effet d'augmenter le nombre de 'hits', et donc le temps d'exécution de l'algorithme. Par conséquent, un compromis est nécessaire, en général T est choisi égal à environ un tiers de la valeur de S. Enfin, le processus d'allongement dans une des directions est arrêté dès que le score de substitution de la paire de symboles que l'on essaye de rajouter aux mots déjà alignés est inférieur à une valeur X fixée par l'utilisateur.

Le programme garde toutes les chaînes de la banque possédant au moins un 'hit' qui a pu être prolongé jusqu'à être contenu dans une paire de mots de score au moins S et attribue à chacune de ces chaînes le score du meilleur prolongement. Il calcule également la 'p-value' correspondant à ce score, c'est-à-dire, la probabilité avec laquelle un score supérieur ou égal aurait pu être atteint sur une chaîne aléatoire de même longueur et de même composition en monomères. Le calcul de cette probabilité n'est pas réalisé de manière empirique par une comparaison avec de l'aléatoire comme cela est le cas pour FastP/N et FastA mais aussi pour d'autres programmes de criblage moins connus [Collins *et al.*, 1988] [Collins and Coulson, 1990], mais est effectué *a priori*, sur la base d'une théorie statistique établie par Karlin [Altschul, 1991] [Karlin and Altschul, 1990]. Ainsi, cette p-value est donnée par la formule :

$$1 - \ exp^{-Knm \ exp^{-\lambda S}}$$

où K est un facteur de normalisation qui dépend uniquement de la matrice de similarité $\mathcal{M}$ utilisée et λ est donné par $\sum_{i,j \in \Sigma} p_i p_j \exp^{\lambda \mathcal{M}(i,j)} = 1$ avc p_i la fréquence d'apparition du monomère i dans la banque.

C'est cette même formule qui est bien sûr employée dans l'autre sens afin d'obtenir la valeur à attribuer à S, et qui doit correspondre à la valeur du plus grand score que peut présenter une MSP relativement à deux chaînes prises au hasard (en général, celle correpondant à une p-value de 10^{-3} ou 10^{-5}). Enfin, le classement des chaînes de la banque est effectué sur les 'p-values' plutôt que sur les scores. L'emploi de ces p-values afin de d'estimer le degré de ressemblance de la chaîne requête avec les chaînes de la banque, ainsi que l'utilisation d'une unité de comparaison ayant pour base des mots (les w-mères) plutôt que des symboles font toute l'originalité de BLAST, et son importance en pratique pour les biologistes. Quant à l'algorithme lui-même, son intérêt principal réside dans l'automate des w-mères dû à Myers [Altschul *et al.*, 1990].

Certaines étapes de BLAST admettent des variantes. Par exemple, dans la phase de prolongement des 'hits', des trous peuvent être permis. Une forme simplifiée de programmation dynamique est alors utilisée pour effectuer ce prolongement, mais il est clair que cela augmente considérablement le temps d'exécution de l'algorithme et n'est pas une variante vraiment utilisée en pratique. Une autre modification possible consiste à créer la liste des mots à partir des chaînes de la banque de protéines et de rechercher ensuite des 'hits' dans la chaîne requête. Bien sûr, cela demande un espace en mémoire considérable, et exige un accès aléatoire à la banque et, encore une fois, est une variante peu employée.

Une extension bien plus intéressante de BLAST, appelée BLAST3, permet de comparer trois chaînes de la banque simultanéement et non plus deux. La comparaison toutefois n'est pas faite directement mais utilise les paires de mots requête-banque ayant un score suffisament élevé (typiquement supérieurs à une valeur $T' < S$) qui auront été repéré au préalable. Ces paires permettent alors d'établir les alignements, toujours sans trous, de triplets de mots, un mot de la requête et deux de la banque appartenant à des chaînes différentes. Ainsi, si le mot v

sur la chaîne requête r est aligné avec les mots $u1$ et $u2$ de deux chaînes $s1$ et $s2$ de la banque, alors le trio v, $u1$ et $u2$ détermine un certain alignement de r, $s1$ et $s2$, que BLAST3 essaye de prolonger de façon similaire au prolongement des paires dans BLAST (le score de trois symboles par lequel ce prolongement est tenté étant donné par un score SP : c'est la somme des scores des trois paires de symboles qu'il est possible d'obtenir à partir du trio de départ). À nouveau ici, le choix de la valeur T' est un compromis entre sensibilité et rapidité. Les triplets de mots de score supérieur à T' sont quant à eux repérés comme le sont ceux de score supérieur à S dans BLAST (c'est-à-dire, en deux phases), ou bien directement à partir de la table indexée ou automate de la première étape de filtrage de BLAST (localisation des 'hits').

Il est plus difficile d'établir la complexité de BLAST (ou de BLAST3) puisque celle-ci va dépendre des valeurs attribuées à tous ces paramètres, S, T, X (et T' dans le cas de BLAST3). Dans la pire des hypothèses (pour $S = T = T' = 0$ et X égal à la valeur maximale de la matrice de similarité), elle est en $O(n.m.M)$ pour BLAST et $O(n.(m.M)^2)$ pour BLAST3 comme pour les algorithmes de programmation dynamique. En pratique, BLAST est cependant environ 10 fois plus rapide que FastA, dont nous avons vu que la complexité est en $O(\frac{n.m.M}{|\Sigma|^k})$.

4.2.2.1.2.2 Les méthodes approchées pour le multiple

Nous allons passer maintenant au cas multiple et présenter dans cette section quatre heuristiques, dont les trois premières ne sont qu'esquissées. Il convient d'observer dès maintenant qu'aucun de ces quatre algorithmes n'admet de trous dans les blocs de similarité qu'ils identifient.

Les deux premières heuristiques que nous décrivons rapidement procèdent en fixant à l'avance la longueur des similarités locales qu'elles vont pouvoir repérer. Notons cette longueur m. Les ensembles de mots similaires sont alors recherchés soit en faisant de la programmation dynamique à N dimensions mais uniquement à l'intérieur d'une fenêtre de longueur $W > m$ [Johnson and Doolittle, 1986], soit en effectuant des comparaisons progressives [Bacon and Anderson, 1986] [Bacon and Anderson, 1990]. Dans ce second cas, les mots similaires sont localisés en réalisant des comparaisons deux à deux toujours. Initialement deux chaînes sont choisies (sur la base de critères non spécifiés dans l'article) et tous les mots similaires de longueur m recherchés, classés suivant leurs scores et puis placés dans une table. Une troisième chaîne est ensuite choisie (à nouveau les critères pour ce choix ne sont pas donnés) et comparée avec la table des mots similaires repérés sur les deux chaînes initiales. Cette comparaison résulte en une nouvelle table, plus réduite que la première, contenant les mots similaires communs aux trois chaînes. Le processus se poursuit avec les chaînes restantes qui sont comparées à tour de rôle à la table des mots similaires présents sur les chaînes examinées auparavant pour produire à la fin une table des mots similaires communs à toutes. La complexité en temps de ces algorithmes est en $O(N.(n-W).W^{N-1})$ pour Johnson et $O(n^2.N^2.m.M)$ où M est la taille de la table (de 'hachage') pour Bacon, n et N la longueur moyenne et le nombre de chaînes respectivement. Un inconvénient majeur de ces deux approches est bien sûr que longueur des similarités locales doit être fixée *a priori*. Par ailleurs, la recherche de ces similarités n'explore pas tout l'espace possible, soit parce qu'elle n'est en fait réalisée qu'à l'intérieur d'une fenêtre de longueur également fixe (Johnson), soit parce qu'elle limite le nombre de chaînes qui sont effectivement comparées entre elles (Bacon). Les mots similaires trouvés par ce dernier dépendent également fortement de l'ordre avec lequel les chaînes ont été choisies pour être comparées.

La troisième heuristique due à Sheridan [Sheridan and Venkataraghavan, 1992] a pour point de départ une recherche des mots similaires communs à deux ou trois chaînes et utilise pour trouver ces mots le cribleur BLAST [Altschul *et al.*, 1990] et sa version étendue BLAST3

[Altschul and Lipman, 1990] décrits précédemment (section 4.2.2.1.2.1). L'extension de Sheridan au cas de plus de 3 chaînes fait appel à une technique de regroupement (en anglais 'clustering') sur une sélection des 'meilleurs' ensembles de deux ou trois mots similaires (ceux ayant un meilleur score total et, dans le cas des triplets, deux à deux) identifiés dans la première phase grâce à BLAST3, mais qui est différente de celle employée par BLAST3, pour mettre ensemble les paires de mots trouvés dans une étape préliminaire. En effet, le critère pour grouper des duplets ou triplets de mots similaires est cette fois que p des mots doivent avoir un préfixe ou un suffixe de longueur au moins m en commun, où m est typiquement choisi égal à 15 et p à 1 (pour le regroupement de deux duplets) ou 2 (pour le regroupement d'un duplet avec un triplet ou de deux triplets). La complexité de l'algorithme n'est pas donnée mais l'emploi de BLAST3 signifie que, dans le pire des cas, elle est en $O(n^3)$.

Enfin, la quatrième heuristique que nous allons décrire plus en détail est le premier algorithme à avoir été inclus dans un programme appelé MACAW [Schuler *et al.*, 1991] (pour simplifier, nous confondrons dorénavant le nom de cet algorithme avec celui du programme). Cet algorithme est très utilisé en pratique et se libère de presque toutes les limitations des heuristiques antérieures. La longueur des similarités locales n'a ainsi pas besoin d'être établie au préalable et les comparaisons sont faites simultanément et non progressivement comme dans le cas de Bacon. Par contre, les trous ne sont toujours pas autorisés dans les ensembles de mots considérés comme similaires. Ceux-ci sont donc soit des mots identiques, soit des mots qui présentent, les uns par rapport aux autres, un certain nombre de substitutions uniquement. L'espace de recherche n'est pas non plus exploré exhaustivement et l'idée d'une progression demeure dans MACAW puisque les mots similaires communs à $p+1$ des chaînes de départ sont construits en utilisant les mots similaires communs à 2 chaînes obtenus dans une étape initiale. Mais toutes les combinaisons deux à deux des N chaînes sont tentées afin d'en extraire les mots communs, ce qui n'est pas le cas pour Bacon. En fait, MACAW est très ambitieux puisqu'il souhaite repérer non seulement les mots similaires présents dans toutes les chaînes, mais aussi ceux qui ne sont présents que dans un sous-ensemble de celles-ci. Il veut donc pouvoir identifier tous les groupes de mots communs, même lorsque ceux-ci ne contiennent que 2 éléments. C'est une caractéristique intéressante de l'algorithme, nous avons en effet vu dans la section 3.3.3 qu'il est important de pouvoir travailler en présence de bruit.

L'idée de l'algorithme est la suivante. Commençons par le deux à deux. MACAW emploie de la programmation dynamique pour comparer deux chaînes localement mais comme il n'autorise pas les trous, nous pouvons voir les calculs effectués comme un simple comptage, pondéré par des scores, des symboles mis face à face lorsqu'une chaîne $s1$ est alignée avec une seconde chaîne $s2$, pour tout glissement possible de $s1$ par rapport à $s2$. Comme dans le cas de FastN/FastP ou FastA et BLAST, ceci signifie que dans le tableau des scores, seul nous intéresse ce qui se passe le long des diagonales puisque celles-ci représentent tous les alignements possibles sans trous des deux chaînes (voir la figure 4.16).

Pour chaque diagonale, MACAW recherche alors les paires de mots similaires définis comme étant ceux qui ont un score supérieur à une certaine valeur seuil t. Ces paires forment ce que les auteurs appellent des blocs de quorum en chaîne 2 (voir la définition 3.3.4). Idéalement, pour rechercher tous les blocs de quorum en chaîne 3 à N de score supérieur à ce seuil (où le score d'un bloc multiple est la somme des scores de toutes les combinaisons possibles de paires de mots du bloc, c'est-à-dire c'est une somme SP), la procédure serait la même. Elle consisterait donc en un examen de toutes les diagonales d'un tableau ayant de 3 à N dimensions. Une telle démarche conduirait cependant à un algorithme dont la complexité serait en $O((n+1)^{N+1})$, ce qui est pire que de la programmation dynamique classique à N dimensions. L'heuristique qui permet alors à MACAW de demeurer en $O(n^2.N^2)$ repose sur le même principle que celui

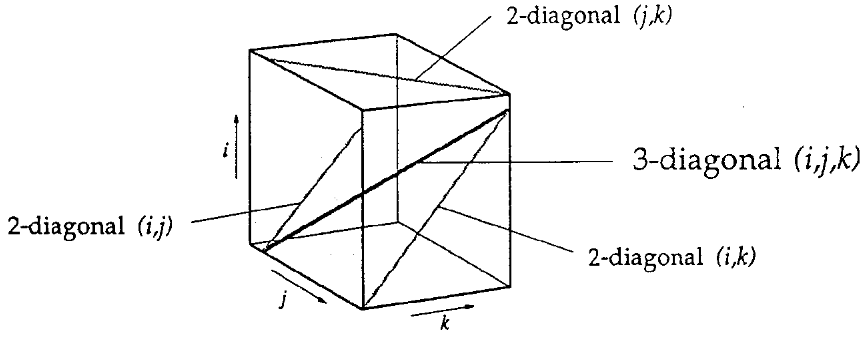


Figure 4.16: Diagonales du tableau des scores dans MACAW.

utilisé par BLAST3, étendu cette fois au cas de N chaînes. En effet, rappelons que les blocs de quorum supérieur ou égal à 2 que MACAW veut retenir sont ceux qui ont un score supérieur à une certaine valeur seuil t. Considérons maintenant un bloc de quorum p strictement supérieur à 2. Il est tout à fait possible (cela sera même souvent le cas) qu'une ou plusieurs des paires de mots de ce bloc n'aient pas un score supérieur à t même si le bloc comme un tout vérifie cette propriété puisque le score d'un bloc est la somme des scores de toutes les paires de mots le composant. L'idée est alors d'imposer une contrainte supplémentaire assez forte qui consiste à exiger qu'un bloc de quorum $p \geq 3$ ait non seulement un score supérieur à t, mais que de plus il soit composé de paires de mots dont chacune a elle-même un score supérieur à t. Ainsi les blocs de quorum $p \geq 3$ sont construits uniquement à partir des blocs de quorum 2 qui auront été retenus dans une première étape de l'algorithme (voir la figure 4.17). MACAW ne recherche donc pas tous les blocs de quorum $p \geq 3$ dont le score serait supérieur à t, mais seulement un sous-ensemble (en pratique souvent considérablement plus petit) de ces blocs. La justification pour une telle contrainte est que les blocs multiples contenant des paires de mots qui ne se ressemblent pas beaucoup ne sont pas intéressants biologiquement. C'est une remarque valable, mais la 'solution' proposée semble un peu drastique dans la mesure où le seuil est fixé à la même valeur quel que soit le quorum du bloc (qu'il comporte donc 2 ou plus de 2 mots) ce qui n'était pas le cas dans BLAST3. On peut se demander si MACAW n'ouvre pas ainsi main en partie de la flexibilité que le multiple permet par rapport au deux à deux (voir la section 3.3.2).

Ce seuil t que doit dépasser le score de tout bloc est déterminé encore une fois sur la base d'un critère statistique, similaire à celui employé par BLAST : c'est le score au-delà duquel un bloc de quorum 2 est en fait considéré comme étant 'significatif', c'est-à-dire comme non dû au hasard. Par ailleurs, une seconde sélection peut être effectuée même sur les blocs de quorum $p > 2$. Ceux-ci ne sont alors retenus que s'ils sont eux-mêmes significatifs. La signification statistique d'un bloc de quorum p quelconque ayant un score égal à $\mathcal{S}$ est une extension au cas p de la théorie pour 2 chaînes et est donnée par la formule [Karlin and Altschul, 1993] [Schuler *et al.*, 1991] :

$$1 - \exp^{-K.n^p.\exp^{-\lambda.\mathcal{S}}}$$

avec ici K et λ calculées d'après les scores possibles pour une p-colonne (c'est-à-dire une colonne ayant p lignes/symboles) et les probabilités correspondant à ces scores. Cette formule suppose que le score moyen d'une colonne est négatif et que les longueurs des p chaînes où se trouve

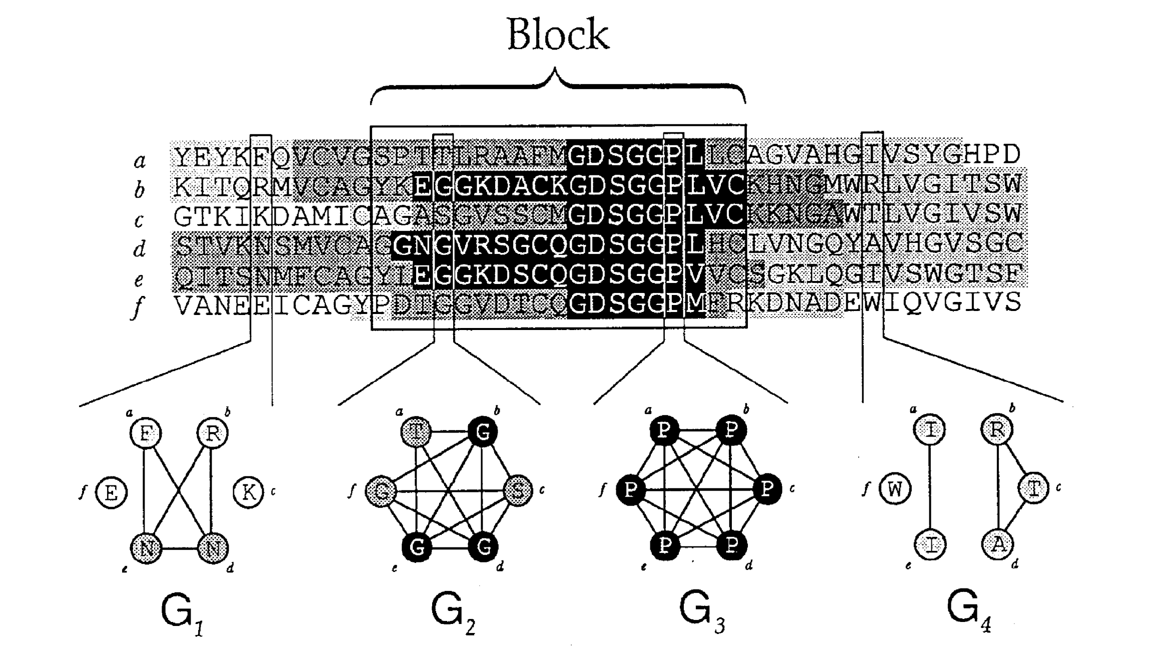


Figure 4.17: Construction des blocs dans MACAW.

le bloc ne sont pas trop différentes. Les blocs de quorum 2 ou plus peuvent ainsi être classés d'après la valeur de cette formule et seuls les plus 'significatifs' conservés. Ce classement permet alors à MACAW de réaliser un alignement multiple des chaînes. Pour cela, il sélectionne le bloc le plus significatif de tous (notez qu'il peut y en avoir plus d'un) et aligne sur lui toutes les chaînes où ce bloc est présent. Puis le second bloc le plus significatif est choisi, et s'il ne présente pas de conflit avec le premier (c'est-à-dire, si ses occurrences ne se situent pas à la fois à gauche et à droite du premier bloc), les chaînes où il est présent sont alignées sur ce second bloc (avec création éventuelle de trous parfois très étendus entre les blocs). Le processus se poursuit tant que cela est possible pour tous les blocs considérés comme étant suffisamment significatifs (voir les figures 4.18 et 4.19 pour l'alignement final).

Il faut cependant noter que le calcul de K et de λ, qui cette fois est fonction du nombre de chaînes et des probabilités correspondant aux scores des colonnes des blocs, est très coûteux. Il est donc en pratique réalisé une fois pour toutes au préalable d'après des scores estimés sur la base des fréquences observées habituellement dans les macromolécules pour chacun des monomères, cela pour un nombre de chaînes variant entre 2 et une valeur maximale qui actuellement est fixée à 16. MACAW ne peut donc pour l'instant repérer des blocs de similarité de manière fiable selon les principes qu'il a lui-même établis que sur un ensemble contenant au plus 16 chaînes.

Le grand intérêt de MACAW est que le programme est suffisamment rapide pour pouvoir être interactif. Un utilisateur peut ainsi éventuellement rejeter un bloc sélectionné par l'algorithme comme étant statistiquement significatif, ou plus simplement l'éditer, c'est-à-dire entre autres, étendre ou diminuer la longueur d'un bloc (i.e. longueur des mots le composant) ou son quorum (i.e. nombre de chaînes que le bloc englobe).

Signalons enfin qu'une technique initialement un peu semblable à celle de MACAW est

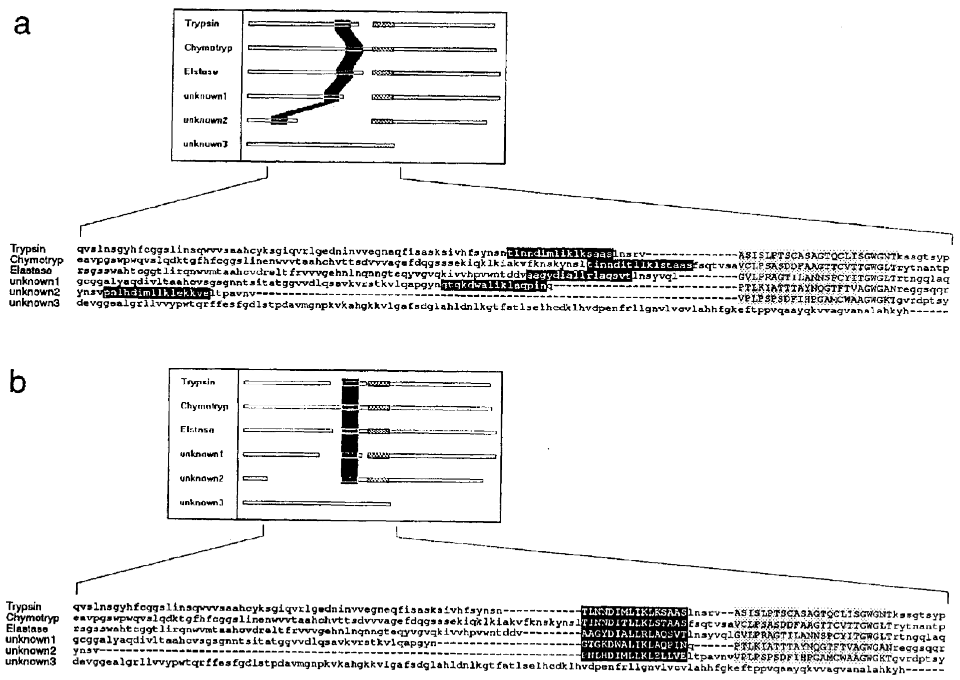
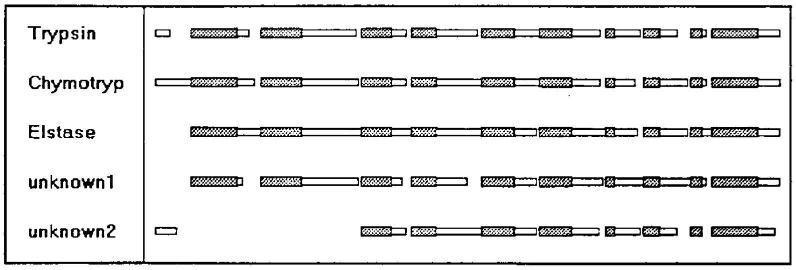


Figure 4.18: Alignement multiple dans MACAW (sur les blocs les plus significatifs).

a



b

```
Trypsin     vddddk---------IVGGYTCGANTVPYQVSLNsgyhf-----CGGSLINSQWVVSAAHCyksgi  52
Chymotryp   cgvpaiqpvlsglsrIVNGEEAVPGSWPWQVSLQdktgfhf---CGGSLINENWVVTAAHCgvtts  63
Elastase    ---------------VVGGTEAQRNSWPSQISLQyrsgsshahtCGGTLIRQNWVMTAAHCvdrel  51
unknown1    ---------------VVGGTRAAQGEFPFMVRLSmg--------CGGALYAQDIVLTAAHCvsgsg  43
unknown2    rkrestqqk---------------------------------------------------------   9

Trypsin     qvrlgedninvvegnwqf--ISASKSIVHPSYNsntlnn--DIMLIKLKSAaslnsrvasislpts  114
Chymotryp   dvvvagefdqgsssekiqk-LKIAKVFKNSKYNsltinn--DITLLKLSTAasfsqtvsavclpsa  126
Elastase    tfrvvvgehnlnqnngteqyVGVQKIVVHPYWNtddvaagyDIALLRLAQSvtlnsyvqlgvlpra  117
unknown1    nntsitatggvvdlqsavk-VRSTKVLQAPGYNgtgk----DWALIKLAQPinqptlkiattta--  102
unknown2    --------------------IKVEKQIIHESYNsvpnlh--DIMLLKLEKKveltpavnvvplpsp  53

Trypsin     ca--SAGTQCLISGWGNTkssgtsypdvLKCLKAPILSDSSCksaypgqitsnm--FCAGyleggk  176
Chymotryp   sddfAAGTTCVTTGWGLTrytnantpdrLQQASLPLLSNTNCkkywgtkikdam--ICAGasgvss  190
Elastase    gtilANNSPCYITGWGLTrtngqlaqt-LQQAYLPTVDYAICssssywgstvknsmVCAGgngvrs  182
unknown1    ----YNQGTFTVAGWGANreggsqqry-LLKANVPFVSDAACrsaygnelvanee-ICAGypdtgg  162
unknown2    sdfiHPGAMCWAAGWGKTgvrdptsyt-LREVELRIMDEKACvdyryyeykfq---VCVGspttlr  115

Trypsin     dscq-GDSGGPVvcsgklq-----GIVSWgs--GCAQKNKPGVYTKVCNYVSwikqtiasn      229
Chymotryp   cm---GDSGGPLvckkngawtlv-GIVSWgs--STCSTSTPGVYARVTALVNwvqqtlaan      245
Elastase    gcq--GDSGGPLhclvnggyavh-GVTSFvsrlGCNVTRKPTVFTRVSAYISwinnviasn      240
unknown1    vdtcqGDSGGPMfrkdnadewiqvGIVSWgy--GCARPGYPGVYTEVSTFASaiasaartl      221
unknown2    aafm-GDSGGPLlcagvah-----GIVSY----GHPDAKPPAIFTRVSTYVPtinavin        164
```

Figure 4.19: Alignement multiple final dans MACAW.

utilisée par Heringa [Heringa, 1994] [Heringa and Argos, 1993] afin de localiser des blocs de mots similaires (avec trous autorisés cette fois). La conservation d'un bloc se fait toutefois désormais sur la base de son score uniquement et il n'y a pas évaluation de la signification statistique des blocs.

4.2.2.2 Méthodes d'optimisation stochastique

Nous ne présentons dans cette section qu'une des méthodes d'optimisation stochastique, celle de Lawrence inspirée par l'approche de Hertz [Hertz *et al.*, 1990] [Cardon and Stormo, 1992] et introduite initialement en 1990 avec l'algorithme EM (en anglais 'Expectation Minimisation') [Lawrence and Reilly, 1990] et puis affinée en 1993 avec la méthode d'échantillonnage de Gibbs [Lawrence *et al.*, 1993]. Nous ne parlerons donc pas de l'autre méthode utilisant des HMM ('Hidden Markov Model') [Krogh *et al.*, 1994]. La raison en est que le principe de base est essentiellement le même. Il s'agit en effet toujours d'extraire une information (localisation d'un signal) d'une masse de données (les chaînes) d'après des critères statistiques (EM dans le cas de Krogh par exemple). Les différences fondamentales entre les deux approches sont dans le type de critère statistique utilisé et dans le fait que Lawrence condense l'information extraite dans une matrice de fréquences, tandis que les méthodes HMM la résument dans des automates stochastiques. Ces différences, en particulier quant au critère utilisé, ne sont pas négligeables, mais il n'est pas essentiel de les détailler dans le cadre de ce travail.

Il est intéressant avant de décrire l'algorithme de Lawrence dans sa seconde version (celle concernant l'échantillonnage de Gibbs [Lawrence *et al.*, 1993]) de poser à nouveau dans ses propres termes le problème que nous cherchons à résoudre tel que celui-ci a été énoncé dans [Lawrence and Reilly, 1990] :

Le problème de Lawrence *Soit un ensemble de N chaînes dont on sait que toutes contiennent un signal biologique de longueur k. Le problème est d'identifier la localisation de ce signal sur chacune des chaînes.*

Observons que dans son énoncé original, la longueur du signal est connue, et que celui-ci est supposé être présent dans chacune des chaînes de l'ensemble. Ces hypothèses ne sont pas vraiment nécessaires mais travailler avec elles simplifie l'explication de l'algorithme, aussi nous allons les accepter pour l'instant.

Si nous reformulons le problème en termes plus familiers d'un bloc, ce qu'on nous demande de faire est localiser l'emplacement du bloc de longueur k et quorum N correspondant au signal porté par un ensemble de N chaînes, c'est-à-dire repérer son occurrence sur chacune de ces chaînes. Bien sûr la réussite d'une telle entreprise suppose que le bloc présente une certaine caractéristique permettant de le différencier de tous les autres blocs que l'on peut construire à partir de l'ensemble des N chaînes.

Cette caractéristique est justement celle qui correspond à la définition 3.4.11 donnée dans le chapitre 3. Le bloc recherché est ainsi celui qui maximize la valeur de :

$$\log L = \sum_{i=1}^{k} \sum_{j \in \Sigma} n_{ij} \log \frac{q_{ij}}{p_j}$$

où Σ est l'alphabet des monomères (nucléotides ou acides aminés), n_{ij} est le nombre de fois où le monomère j apparaît dans la colonne i de l'alignement sans trous des mots, q_{ij} est la fréquence d'apparition du monomère j à cette position et p_j est la fréquence d'apparition du monomère j sur les chaînes ailleurs que dans les mots du bloc. Rappelons que q_{ij} et p_j sont éventuellement corrigés par des régularisateurs. Comment procède l'algorithme pour trouver ce bloc? La façon exacte, et naïve de faire consisterait à calculer la valeur de L pour tous les blocs possibles de quorum N et longueur k et voir lequel maximize cette valeur. Une telle approche serait en $O(n^N)$. L'heuristique de Lawrence consiste à choisir initialement un emplacement quelconque pour chaque mot du bloc correspondant au signal recherché, notons ce bloc B, sur chacune des chaînes de l'ensemble. L'algorithme procède alors en alternant une étape prédictive avec une étape d'échantillonnage de la façon suivante :

- étape prédictive : une des chaînes est sélectionnée, soit au hasard, soit l'une après l'autre dans un ordre fixé à l'avance. Notons cette chaîne s. Les descripteurs n_{ij}, q_{ij} et p_j sont calculés (éventuellement corrigés) d'après les positions actuelles des mots du bloc B sur toutes les chaînes à l'exception de s (cela signifie que le 'contenu' du bloc est figé par rapport à toutes les chaînes sauf s);

- étape d'échantillonnage : tous les mots possibles de longueur k sur la chaîne s sont alors considérés à tour de rôle comme candidats pour venir s'intégrer au bloc B et la valeur de la formule ci-dessus est établie par rapport à chacun de ces mots x sur s (rappelons que les mots sur les autres chaînes demeurent les mêmes). Notons cette valeur L_x. Un de ces mots est alors choisi avec une probabilité $\frac{L_x}{\sum_x L_x}$ pour venir se joindre au bloc B.

Les deux étapes sont répétées jusqu'à convergence, c'est-à-dire, jusqu'à ce que les positions des mots du bloc sur chacune des chaînes ne bouge plus d'une itération à l'autre, ou jusqu'à ce que la différence entre deux valeurs consécutives pour la formule devienne suffisamment petite pour que l'on considère l'apport d'information à partir de ce point statistiquement négligeable. L'idée derrière cet algorithme est que l'identification de l'emplacement correct du signal même sur une seule des chaînes de départ augmente le pouvoir qu'ont les descripteurs n_{ij}, q_{ij} et p_j de repérer ce signal sur les autres chaînes, et l'augmente d'autant plus (i.e. les descripteurs deviennent d'autant plus discriminants) que l'algorithme a progressé dans son déroulement. On peut comparer ce processus à celui de certaines formes d'apprentissages qui commencent lentement puis s'accélèrent à partir du moment où une certaine masse de connaissances a pu être accumulée de telle sorte que ce qui est appris 'à la fin' est assimilé beaucoup plus rapidement que ce qui doit être compris au départ. L'image d'un apprentissage dans ce cas précis est également appropriée dans la mesure où le choix d'un emplacement sur une des chaînes n'est jamais accompli de manière définitive. Ainsi un mauvais choix effectué à une étape donnée peut se trouver corrigé à une étape ultérieure lorsque le modèle (le bloc B) est devenu plus 'proche' du signal recherché. La convergence est assurée néanmoins car les choix ne sont pas non plus réalisés complètement au hasard. Observons que cela n'est pas vrai pour la version de 1990 ('Expectation Minimisation'). Pour cette version en effet, l'emplacement optimal du bloc sur chacune des chaînes n'est choisi qu'à la fin. L'étape d'échantillonnage est ainsi remplacée par une étape d'estimation qui calcule la matrice des fréquences de chaque monomère j, pour chaque position du bloc, lorsque celui-ci est placé sur tous les mots x de la chaîne, pondérées par la valeur L_x. Le 'bon' fonctionnement de l'algorithme dépend alors fortement du choix de la matrice initiale qui ne peut plus être réalisé de manière tout à fait aléatoire si l'on veut éviter les optimaux locaux. Nous verrons une illustration de cela dans le chapitre 6. La méthode d'échantillonnage de Gibbs a été élaborée précisément pour résoudre ce problème.

Ainsi que nous l'avons dit, ni la longueur du bloc (i.e. la longueur des mots le composant) ni son quorum n'ont besoin d'être fixés à l'avance. L'algorithme est capable également de rechercher plusieurs blocs simultanément, à condition toutefois que ceux-ci ne présentent pas de conflit entre eux. Lawrence ne peut rechercher que les trains de blocs, pas les blocs dont les occurrences se croiseraient sur les chaînes. Une des façons possibles de procéder consiste à incorporer l'espacement entre les blocs directement dans la formule caractérisant la quantité d'information que ces blocs représentent [Cardon and Stormo, 1992]. L'algorithme de Lawrence a une complexité en temps qui est $O(T.n.k.N)$ où k est la longueur du bloc et T est un facteur qui est proportionnel à n mais, en pratique, inversément proportionnel à N (l''apprentissage' se fait d'autant plus rapidement que le nombre de chaînes — i.e. le nombre d'exemples — est grand). La complexité en espace est $O(n.N)$.

Les algorithmes de recherche de mots communs par optimisation stochastique sont très efficaces en général, du point de vue aussi bien de leur temps d'exécution que des résultats qu'ils produisent. Ils représentent également une approche qui est surprenante sous bien des aspects. Nous pouvons illustrer la méthode de Lawrence en particulier par l'image de quelqu'un qui déterminerait l'emplacement d'un objet, un bloc de mots, dans un certain espace, un ensemble de chaînes, en jouant aux dés et en se servant pour cela de dés qui auraient été convenablement 'pippés'. C'est une méthode tout à fait valable (et validée par la qualité des résultats obtenus), et de plus fondée sur une théorie mathématiquement justifiable. Bien sûr, la théorie s'appuie sur un modèle statistique qui simplifie sans doute la réalité biologique, mais qui est admissible en première approximation. Ce qui peut paraître plus troublant par contre, c'est que si cette approche permet de détecter des signaux, elle ne donne aucune information sur les mécanismes potentiels associés à ces signaux. Ceci vient du fait qu'elle se base sur une fonction du contenu global du signal et non sur la structure fine de celui-ci.

4.2.3 Méthodes combinatoires exactes

4.2.3.1 Introduction

Afin de compléter ce chapitre consacré aux méthodes directes de la ressemblance qui est aussi un chapitre de présentation du travail accompli par d'autres que nous, nous devrions maintenant décrire toutes les approches combinatoires exactes qui permettent d'identifier des blocs de mots communs, identiques ou similaires, dans un ensemble de chaînes par une comparaison de ces mots entre eux. Ces approches utilisent la relation de similarité entre monomères donnée dans la définition 3.4.10 du second chapitre et peuvent être divisées en deux grandes catégories. La première englobe toutes les méthodes travaillant avec une relation R de similarité qui est en fait l'identité ou une relation d'équivalence, et la seconde pouvant employer une relation symétrique, réflexive mais non transitive quelconque. Dans le premier cas, les blocs de mots communs trouvés forment des classes de mots identiques ou équivalents, dans le second, ils forment des cliques maximales de la relation entre monomères étendue aux mots. En termes d'algorithmes, les méthodes combinatoires directes et exactes peuvent également être divisées en trois autres groupes différents des précédents, le premier utilisant des techniques simples d'indexation pour localiser ces blocs de similarité, la seconde se servant de structures particulières de données comme les arbres ou les automates des suffixes (ces deux premiers groupes sont entièrement inclus dans celui des méthodes pour lesquelles R est l'identité ou une relation d'équivalence), et enfin le troisième employant des techniques d'attribution d'étiquettes qui sont aussi des techniques de partitionnement ou de raffinement d'ensembles pouvant être réalisées sans stocker autant d'information que les deux premières approches.

Comme ces dernières sont à la base des méthodes indirectes de comparaison de chaînes que nous avons élaborées dans le cadre de ce travail, nous allons donc nous contenter ici d'en fournir l'idée essentielle sous un aspect seulement — celui d'une attribution d'étiquettes — en laissant pour le chapitre 5 une discussion plus approfondie de leurs autres caractéristiques. Avant cela nous présentons les algorithmes de recherche de mots identiques par indexation simple ou utilisant une arbre ou un automate des suffixes.

4.2.3.2 Recherche de mots communs identiques par indexation simple

Commençons par revenir un peu en arrière pour rappeler l'algorithme de MACAW que nous avons vu précédemment et qui permet de repérer des blocs de mots similaires sans trous dans un ensemble de chaînes. Supposons que les blocs recherchés par MACAW aient été en fait des blocs de mots identiques (il suffit pour cela de fixer le score de substitution d'un monomère par lui-même à une valeur supérieure au seuil t et par un autre monomère à $-\infty$). MACAW aurait besoin de réaliser $O(n^2.N^2)$ opérations pour trouver ces blocs. Cela paraît excessif.

Considérons maintenant des algorithmes comme ceux de Leung [Karlin *et al.*, 1988a] [Karlin *et al.*, 1988b] [Leung *et al.*, 1991], de Martinez [Martinez, 1983] [Martinez, 1988] [Sobel and Martinez, 1986] ou de Santibanez [Santibanez and Rohde, 1987] qui identifient les blocs de mots identiques en codant ces mots sur des valeurs entières qui leur servent d'indexes. Ces mots indexés sont soit placés dans une table [Karlin *et al.*, 1988a] [Karlin *et al.*, 1988b] [Leung *et al.*, 1991] ce qui permet de les classer en un temps $O(n.N)$, soit triés dans des listes [Martinez, 1983] [Martinez, 1988] [Santibanez and Rohde, 1987] [Sobel and Martinez, 1986] qui peuvent être construites et manipulées en un temps $O(n.N.\log(n.N))$. Dans le premier cas, la longueur des mots communs repérables initialement est limitée par la taille de la table et ne peut être prolongée qu'à travers l'usage de listes chaînées ce qui ramène la complexité à $O(n.N.\log(n.N))$ comme pour les algorithmes de Martinez et Santibanez, mais est moindre cependant que celle de MACAW. Bien sûr, MACAW peut accepter des substitutions dans les blocs sans augmenter sa complexité algorithmique. Les trois algorithmes mentionnés ici permettent aussi des substitutions, dans une bien moindre mesure cependant, puisque la relation qui joue entre les symboles (et donc les mots) n'a pas besoin d'être l'identité mais peut être simplement une relation d'équivalence. L'algorithme de Leung admet également des substitutions dans un sens plus général, ainsi que des trous, mais uniquement entre les blocs de mots identiques identifiés au préalable et à condition que les régions d'erreurs ne dépassent pas une certaine longueur. La caractérisation du résultat final obtenu est alors impossible à formuler de manière précise aussi l'algorithme avec ces régions d'erreurs ne peut plus être qualifié d'exact. En fait, nous pouvons voir ces blocs de mots identiques initialement repérés par Leung, ainsi que ceux trouvés par Martinez et Santibanez, comme des points d'ancrage pour un alignement multiple *a posteriori* des chaînes de départ de façon similaire à ce qui est fait dans MACAW. Les critères utilisés pour sélectionner et éventuellement classer les blocs sont toutefois différents en ce qui concerne les algorithmes de Martinez et Santibanez puisque dans leur cas ces critères ont pour base des scores et non une évaluation statistique.

Observons que la classification des algorithmes de cette section parmi les méthodes exactes est un peu contestable. Elle n'est correcte que si l'on s'arrête pour chacun d'eux à la construction des blocs de mots communs identiques ou équivalents. Les prolongements de ces blocs de manière inexacte par Leung ou l'utilisation des blocs comme points fixes entre lesquels aligner des régions moins conservées [Martinez, 1983] [Martinez, 1988] [Santibanez and Rohde, 1987] [Sobel and Martinez, 1986] les situe parmi les heuristiques soit locales, soit globales respectivement.

4.2.3.3 Recherche de mots communs identiques utilisant un arbre ou un automate des suffixes

Bien que n'ayant pas été initialement conçus dans ce but, les arbres des suffixes [Chen and Seiferas, 1985] [McCreight, 1976] [Ukkonen, 1992] [Ukkonen and Wood, 1993] [Weiner, 1973] et les automates des suffixes appelés DAWGs ('Directed Acyclic Word Graph') [Blumer *et al.*, 1985] [Blumer *et al.*, 1983] [Crochemore, 1985] [Crochemore, 1986] peuvent être utilisés pour trouver soit le plus long mot (facteur) commun à un ensemble de chaînes, soit tous les mots communs de longueur 2 à la longueur maximale où par mots communs on entend des mots identiques (ou équivalents moyennant une relation d'équivalence sur l'alphabet et une réécriture des chaînes). Une telle utilisation a déjà été effectuée dans le cadre soit de la recherche de mots répétés de manière contiguë dans une seule chaîne (ce sont les mots carrés ou qui sont puissances d'un même mot primitif — voir en particulier Apostolico [Apostolico and Preparata, 1983] et [Kosaraju, 1994] qui travaillent tous deux avec un arbre des suffixes), soit de la recherche de mots répétés non nécessairement contigus dans une ou plusieurs chaînes (Gusfield [Gusfield, 1995] et Hui [Hui, 1992] qui se servent d'un arbre des suffixes pour cela, Clift qui emploie un DAWG [Clift *et al.*, 1986]).

Les arbres aussi bien que les automates des suffixes réalisent en fait une indexation d'une ou de plusieurs chaînes qui est plus compacte que les indexes établis avec des tables ou des listes chaînées, plus efficaces à utiliser dans la plupart des applications, et permettent de révèler la structure interne des chaînes. Il n'est pas notre propos de détailler ici leur mode de construction, les personnes intéressées peuvent consulter les articles mentionnés plus haut ou le livre de M. Crochemore [Crochemore and Rytter, 1994], mais seulement de donner une idée très succinte de ce que représentent ces deux structures et comment les utiliser pour résoudre le problème qui nous concerne : localiser les blocs de mots communs (ici identiques) d'un ensemble de chaînes.

Considérons d'abord le cas le plus simple d'un arbre des suffixes pour une seule chaîne $s = s_1 ... s_n$. Un arbre des suffixes A pour s est un arbre orienté ayant une racine r et exactement n feuilles numérotées de 1 à n. Chaque nœud interne de l'arbre autre que r possède au moins deux fils et chaque branche de l'arbre est étiquetée par un mot non vide de s sachant que deux branches issues d'un même nœud ne peuvent recevoir des étiquettes commençant avec le même symbole. La caractéristique principale de l'arbre A est que, pour toute feuille i, la concaténation des étiquettes des branches formant le chemin de r à i épelle exactement le suffixe $s_i ... s_n$ de s. Un exemple de l'arbre des suffixes pour la chaîne $s = aabbabd$ est donnée dans la figure 4.20. Bien sûr, pour des raisons d'espace, les branches ne sont pas vraiment étiquetées par des mots mais plutôt par une paire d'entiers indiquant les positions de début et fin du mot dans s (voir la figure 4.21).

La définition d'un arbre des suffixes telle que donnée ci-dessus ne garantit pas que celui-ci existe vraiment pour toute chaîne s. En effet, si un des suffixes de s est un préfixe d'un autre suffixe de s (par exemple, la chaîne $s' = aabbab$ possède un suffixe ab qui est préfixe du suffixe $abbab$ de s'), alors aucun arbre des suffixes correspondant à la définition ne serait possible puisque le chemin dans l'arbre relatif au premier suffixe n'aboutirait pas à une feuille. Pour éviter ce problème, nous devons donc supposer que le dernier symbole de s apparaît nulle part ailleurs dans s. Il suffit pour cela d'ajouter un symbole spécial à la fin de la chaîne s qui n'appartienne pas à l'alphabet sur lequel s est défini. Un arbre des suffixes pour une chaîne s de longueur n peut être construit en un temps $O(n)$ et occupe un espace également $O(n)$ [McCreight, 1976].

Étant donné un tel arbre pour une chaîne s, les mots répétés de s correspondent alors aux étiquettes concaténées des chemins dans l'arbre qui commencent à la racine et se terminent dans un des nœuds internes (voir la figure 4.23). Ils peuvent être énumérés par un simple parcours en profondeur dans l'arbre. Le quorum des blocs qu'ils forment est donné par le nombre de feuilles du sous-arbre ayant le nœud interne pour racine.

L'arbre des suffixes de $N \geq 2$ chaînes est quant à lui construit de manière très similaire à l'arbre pour une chaîne, il suffit simplement de distinguer les chaînes entre elles en concaténant

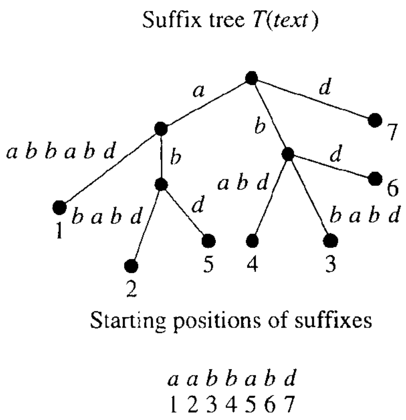


Figure 4.20: Arbre des suffixes pour la chaîne *aabbabd* — exemple pris dans le livre de M. Crochemore [Crochemore and Rytter, 1994].

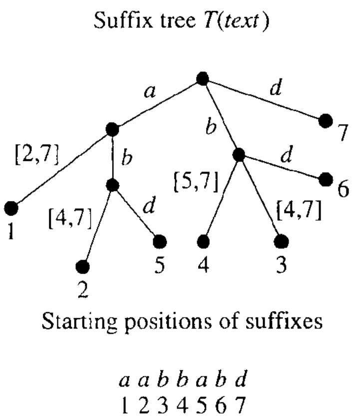


Figure 4.21: Arbre des suffixes pour la chaîne *aabbabd* : étiquetage des branches par des paires d'entiers — exemple pris dans le livre de M. Crochemore [Crochemore and Rytter, 1994].

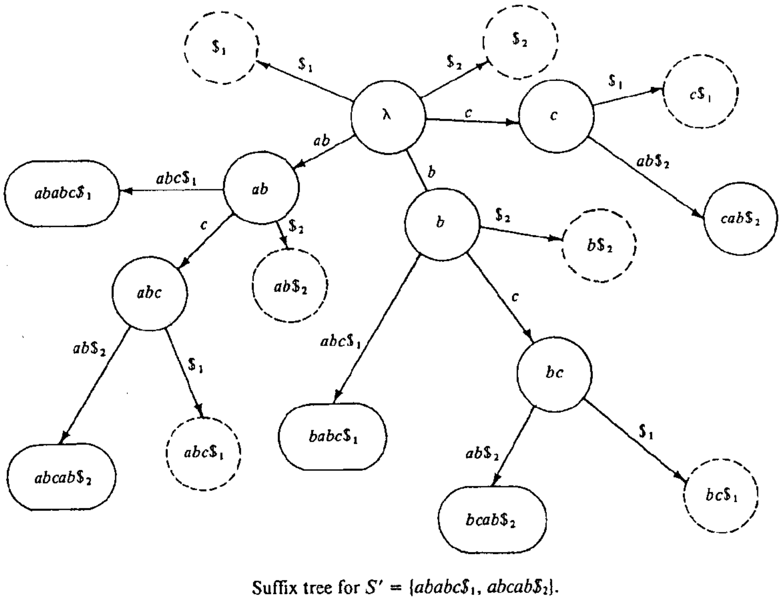


Figure 4.22: Arbre des suffixes pour les chaînes *ababc* et *abcab*.

à la fin de chacune un symbole spécial non présent dans l'alphabet et distinct des symboles spéciaux accolés aux autres chaînes (voir la figure 4.22). Cet arbre possède $n.N$ feuilles si l'on considère des chaînes de même longueur et sont numérotées de $(1,1)$ à (N,n). Étant donné une feuille (p,i), la concaténation des étiquettes des branches formant le chemin de la racine r à (p,i) épelle cette fois le suffixe $sp_i...sp_n$ de la chaîne sp de l'ensemble. Gusfield propose alors un algorithme qui utilise l'arbre des suffixes de $N \geq 2$ chaînes pour trouver le plus long bloc de mots identiques satisfaisant la contrainte de quorum q pour q allant de 2 à N en un temps $O(q.n.N)$, réduit ensuite à $O(n.N)$ par Hui [Hui, 1992]. Notons que le quorum dans le cas de N chaînes correspond bien sûr au nombre de feuilles se rapportant à des chaînes distinctes.

La différence fondamentale entre un arbre des suffixes et un DAWG est que le DAWG est obtenu en identifiant les sous-arbres isomorphiques de l'arbre non compacté A' représentant les suffixes d'une (ou de plusieurs) chaîne s. A' est l'arbre A donné précédemment (figure 4.20), où chaque branche étiquetée par un mot de longueur l supérieure à 1 est remplacée par l branches étiquetées par un unique symbole de l'alphabet (voir la figure 4.24). Certains nœuds internes de l'arbre A' peuvent n'avoir qu'un seul fils. Le DAWG correspondant est donné dans la figure 4.25, il constitue ainsi une forme compactée de l'arbre A' différente de celle obtenue par l'arbre des suffixes. Dans le cas d'une seule chaîne, chaque nœud du DAWG représente la classe d'équivalence de tous les mots ayant mêmes positions finales dans s. Ces positions et/ou leur nombre peuvent être aisément stockés dans les nœuds du DAWG lors de la construction de ce dernier et les mots répétés dans s correspondent alors aux étiquettes concaténées de tous les chemins allant de la source jusqu'à un nœud pour lequel ce nombre est supérieur à 1. Un avantage du DAWG par rapport aux arbres des suffixes est que toute arête de l'automate est étiquetée par un seul symbole. C'est donc une structure plus pratique à utiliser lorsque l'information qui nous intéresse est celle associée aux arêtes plutôt qu'aux nœuds.

La construction d'un DAWG demande le même temps que celle d'un arbre des suffixes et

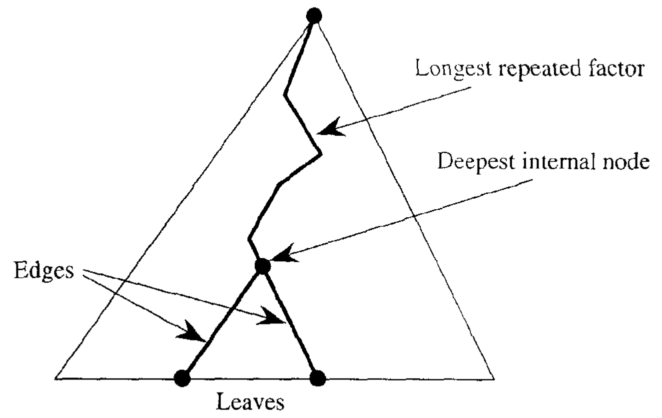


Figure 4.23: Recherche du plus long mot (facteur) dans une chaîne avec un arbre des suffixes — exemple pris dans le livre de M. Crochemore [Crochemore and Rytter, 1994].

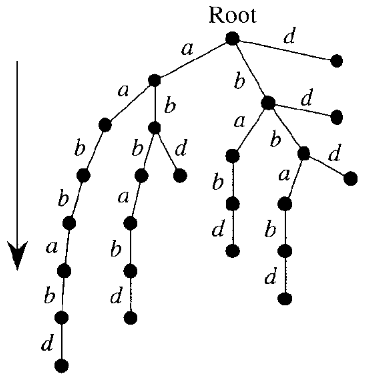


Figure 4.24: Arbre des suffixes non compacté pour la chaîne *aabbabd* — exemple pris dans le livre de M. Crochemore [Crochemore and Rytter, 1994].

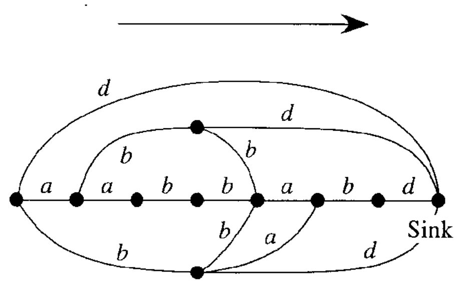


Figure 4.25: DAWG pour la chaîne *aabbabd* — exemple pris dans le livre de M. Crochemore [Crochemore and Rytter, 1994].

l'espace occupé est également proportionnel à la longueur de la chaîne (ou de toutes les chaînes s'il y en a plus d'une). De même que l'arbre des suffixes, le DAWG a été employé par Clift *et al.* [Clift *et al.*, 1986] notamment pour trouver tous les mots répétés dans une séquence d'ADN dans le cadre spécifique cette fois d'une application en biologie.

4.2.3.4 Recherche de mots communs identiques ou similaires par attribution d'étiquettes

Considérons le cas d'une seule chaîne s, et la recherche de mots répétés de manière identique dans s. Que pouvons-nous dire de la composition d'un mot de longueur $k > 1$ — notons-le u — qui se trouve présent au moins deux fois dans s à des positions différentes, par exemple i et j? Notons v et w les deux mots non vides de longueurs k' et k'' respectivement tels que $u = vw$. Nous pouvons alors affirmer que v et w sont aussi des mots répétés au moins deux fois dans s, en particulier v est présent aux positions i et j et w aux positions $i + k'$ et $j + k'$. Tout mot répété de longueur $k > 1$ est ainsi toujours composé de deux mots répétés juxtaposés dans s de longueurs $k' \geq 1$ et $k'' \geq 1$ telles que $k' + k'' = k$.

Cette idée simple forme la base d'un algorithme élaboré par Karp, Miller et Rosenberg en 1972 [Karp *et al.*, 1972] pour trouver tous les mots répétés de façon identique dans une chaîne et que nous appelerons dorénavant KMR. Elle est formellement présentée dans le lemme donné ci-dessous et illustrée dans la figure 4.26. E représente une relation d'équivalence sur Σ qui, dans le cas de KMR, est l'identité I.

Lemme 4.2.1 *Soient* $s \in \Sigma^n$, $a, b \in \{1, ..., n\}$ *et* $i, j \in \{1, ..., n - (a + b) + 1\}$. *Alors :*

$$i \; E_{a+b} \; j \text{ si et seulement si } i \; E_a \; j \text{ et } i + a \; E_b \; j + a.$$

La décomposition d'un mot u peut en fait s'effectuer de plusieurs façons. Ainsi, soient z et t deux mots non vides distincts de v et w et tels que $u = vw = zt$, et supposons sans perte de généralité que $\mid v \mid > \mid z \mid$: nous pouvons alors affirmer que u est un mot présent au moins deux fois dans s si et seulement si v et t sont eux-mêmes des mots répétés. Cette observation forme le second lemme dont KMR se sert pour trouver tous les mots répétés dans une chaîne s. Ce lemme est donné ci-dessous et illustré dans la figure 4.27.

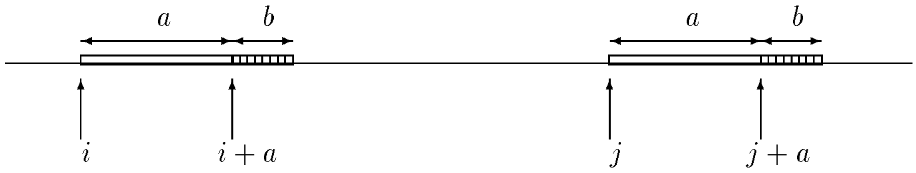


Figure 4.26: Une illustration du premier lemme de KMR.

Figure 4.27: Une illustration du second lemme de KMR.

Lemme 4.2.2 *Soient $s \in \Sigma^n$, $a, b \in \{1, ..., n\}$ avec $b \leq a$ et $i, j \in \{1, ..., n - (a+b) + 1\}$. Nous avons alors :*

$$i\ E_{a+b}\ j \text{ si et seulement si } i\ E_a\ j \text{ et } i+b\ E_a\ j+b.$$

Notons que, lorsque a est égal à b, les deux lemmes sont bien sûr identiques.

Une fois ces lemmes établis qui nous fournissent un moyen de repérer les mots répétés par récurrence, il y a plusieurs manières possibles de se représenter le fonctionnement actuel de l'algorithme. Nous pouvons le voir comme un algorithme de partitionnement d'ensembles (les classes d'équivalence de la relation d'identité entre symboles étendue aux mots), ou comme un parcours dans un arbre qui est étroitement lié à l'arbre des suffixes vu dans la section précédente (ce parcours est imaginaire dans la mesure où aucune structure n'est effectivement construite). Nous pouvons le voir également comme une technique d'attribution d'étiquettes. Il est intéressant d'explorer la question sous tous ces aspects, et nous le ferons en détail en ce qui concerne les deux premiers dans le prochain chapitre. Nous considérons ici uniquement le côté recherche de mots répétés comme une attribution d'étiquettes (numériques). Nous montrons que l'algorithme se réduit alors à un problème de tri. La présentation que nous en faisons est celle donnée par Crochemore [Crochemore and Rytter, 1994].

Considérons les mots répétés de longueur $k \geq 1$ d'une chaîne s et supposons qu'il existe q mots distincts dans s ayant cette longueur. Soit mot(1), mot(2),..., mot(q) la séquence de ces q mots en ordre lexicographique. Étant donné une position i dans s, nous établissons alors que :

$$\text{étiq}_k(i) = j \iff s_i...s_{i+k-1} = \text{mot}(j).$$

La définition de ressemblance (ici l'identité) donnée dans le chapitre 3 (définition 3.4.10) devient :

$$i\ E_k\ j \iff \text{étiq}_k(i) = \text{étiq}_k(j).$$

L'algorithme de KMR consiste alors simplement en un calcul du tableau étiq_k pour tout k.

D'une façon générale, ce calcul se fait par application des lemmes. Supposons étiq_a et étiq_b connus, c'est-à-dire, supposons connues les deux listes :

$$\mathcal{L}_a = (étiq_a(1), étiq_a(2), ..., étiq_a(n-a+1))$$

$$\mathcal{L}_b = (étiq_b(1), étiq_b(2), ..., étiq_b(n-b+1))$$

(dans le cas du lemme 4.2.2, il suffit seulement en fait de connaître la liste $\mathcal{L}_a$). Voyons maintenant comment calculer étiq_{a+b} et $\mathcal{L}_{a+b}$. Soit la liste $\mathcal{L}'_{a+b}$ des étiquettes possibles pour les mots de longueur $a+b$ de s donnée par :

$$\mathcal{L}'_{a+b} = (étiq_a(1)étiq_b(1+a), 1), (étiq_a(2)étiq_b(2+a), 2), ...,$$
$$(étiq_a(n-a-b+1)étiq_b(n-b+1), n-a-b+1))$$

ou bien par :

$$\mathcal{L}'_{a+b} = (étiq_a(1)étiq_a(1+b), 1), (étiq_a(2)étiq_a(2+b), 2), ...,$$
$$(étiq_a(n-a-b+1)étiq_a(n-a+1), n-a-b+1))$$

où chaque étiquette de mot est cette fois associée avec la position dans s où ce mot apparaît. Une fois cette liste triée en ordre lexicographique sur le premier champ et partitionnée en classes ayant ce premier champ identique, il suffit de renuméroter ces classes à partir de 1 pour obtenir les étiquettes étiq_{a+b} des classes des mots identiques de longueur $a+b$. La liste $\mathcal{L}_{a+b}$ donnée par :

$$\mathcal{L}_{a+b} = (étiq_{a+b}(1), étiq_{a+b}(2), ..., étiq_{a+b}(n-a-b+1))$$

est quant à elle obtenue en réordonnant selon son second champ cette fois (indiquant les positions dans s) la liste $\mathcal{L}'_{a+b}$ préalablement triée de manière lexicographique.

Le coût de ces opérations se réduit à celui d'un tri lexicographique sur au plus n éléments de l'intervalle entier (1,2,...,n) qui peut être réalisé en un temps $O(n)$ [Aho *et al.*, 1983], plus celui d'un second tri sur les positions qui peut être réalisé également en $O(n)$ à l'aide d'un tableau auxiliaire de taille n. Si nous faisons $a = b$, l'algorithme KMR requiert donc $O(n. \log k)$ opérations pour repérer les blocs de mots identiques d'une chaîne s quelle que soit la longueur k du bloc ainsi qu'illustré dans la figure 4.28. Observons que k est au maximum égal à $n-1$.

Dans le cas où la longueur k des mots communs recherchés n'est pas une puissance de 2, on utilise un des deux lemmes avec $a = b$ jusqu'à la plus grande puissance de 2 inférieure à la longueur désirée, puis on recourt au lemme 2 avec $a \neq b$ afin de calculer nom_k. La recherche des plus longs blocs est effectuée de manière similaire. Les noms des blocs sont attribués toujours en doublant la longueur de ceux-ci à chaque étape, mais, cette fois, jusqu'à la plus petite puissance de 2 qui est supérieure à la longueur k_{max} pour laquelle il existe encore au moins un bloc de mots communs possédant au minimum deux éléments. La recherche des blocs de longueur notée k_{max} procède ensuite par dichotomie comme indiqué dans la figure 4.28.

L'extension au cas de N chaînes $s1, s2, ..., sN$ avec une contrainte de quorum a été réalisée par Landraud [Landraud *et al.*, 1989] et est immédiate. Les N chaînes sont concaténées en une

/* Recherche de tous les blocs de mots répétés identiques de longueur k. */

construire le tableau étiq_1;
soit d la plus petite puissance de 2 inférieure à k $(d = 2^{\lfloor \log_2 k \rfloor})$
appliquer le lemme 4.2.1 ou 4.2.2 avec $a = b$ pour construire les tableaux étiq_2, étiq_4, étiq_8,..., étiq_d;
si $k = d$
    arrêt;
sinon
    utiliser le lemme 4.2.2 pour construire étiq_k à partir de étiq_d.

/* Recherche de tous les blocs des plus longs mots répétés identiques. */

construire le tableau étiq_1;
appliquer le lemme 4.2.1 ou 4.2.2 avec $a = b$ pour construire les tableaux d'étiquettes étiq_2, étiq_4, étiq_8,..., étiq_d où d est la plus petite puissance de 2 pour laquelle les blocs obtenus possèdent encore au moins deux éléments;
réaliser une recherche dichotomique pour trouver les plus longs blocs :
    $\text{étiq}_{\frac{d}{2}}$ est utilisé pour construire $\text{étiq}_{\frac{3d}{4}}$ (lemme 4.2.2)
    si au moins une étiquette dans $\text{étiq}_{\frac{3d}{4}}$ correspond à un bloc possédant au minimum deux éléments
        utiliser $\text{étiq}_{\frac{3d}{4}}$ pour construire $\text{étiq}_{\frac{7d}{8}}$;
    sinon
        utiliser $\text{étiq}_{\frac{d}{2}}$ pour construire $\text{étiq}_{\frac{5d}{8}}$ etc.

Figure 4.28: Schéma de l'algorithme KMR.

seule — notons-la encore s — et les blocs de mots identiques sont repérés dans cette chaîne $s = s1s2...sN$ en prenant soin de ne pas permettre qu'un mot répété dans un bloc appartienne en fait à deux chaînes différentes. Il suffit pour cela de placer entre les chaînes $s1, s2, ..., sN$ dans s un symbole spécial non présent dans l'alphabet. Ainsi s est en réalité égal à $s1\$s2\$...\$sN$ avec $\$ \notin \Sigma$. En conservant les positions de début de chacune des chaînes originales dans s, la vérification de la contrainte de quorum demande un simple parcours du tableau des noms et se réalise en un temps $O(n.N)$. La complexité en temps finale de l'algorithme est en $O(n.N.\log k)$ où k est au maximum égal à n. L'espace requis est $O(n.N)$.

La substitution d'une relation d'identité ou d'équivalence par une relation quelconque non transitive R ne change presque rien au fonctionnement de l'algorithme proprement dit. Les deux lemmes demeurent valables, E est simplement remplacé par R.

Les blocs repérés ne représentent toutefois plus des classes de mots identiques ou équivalents, mais des cliques de mots tous en relation deux à deux par R_k. Comme R est non transitive, un mot de longueur k peut également appartenir à plus d'une clique et avoir ainsi plus d'une étiquette. En fait, un mot possède entre un et g^k étiquettes, où g mesure le degré de non-transitivité de R (voir la section 3.2.3.2 du chapitre 3). La liste $\mathcal{L}'$ des étiquettes de longueur $k = (a + b)$ donnée précédemment pour $R = E$ devient ainsi une liste plus étendue, où une même position peut apparaître jusqu'à g^k fois comme second champ d'un élément. Le tri lexicographique de $\mathcal{L}'$ se fait de la même façon, ainsi que la séparation de la liste triée en groupes ayant un premier champ identique. Cette séparation demeure un partitionnement par rapport à ce premier champ, mais ne l'est plus par rapport au second dénotant les positions des mots dans s. Il suffit ensuite de numéroter ces groupes à partir de 1 pour obtenir le tableau étiq_k des cliques de longueur k.

Le processus de construction des tableaux étiq_k pour un k quelconque, ou du tableau $\text{étiq}_{k_{max}}$, et donc des cliques est semblable à celui adopté par KMR et indiqué dans l'algorithme de la figure 4.28. La longueur des mots similaires repérés est ainsi doublée à chaque étape tant que cela est possible. La complexité en temps de cette opération pour le cas de $N \geq 1$ chaînes de longueur moyenne n est $O(n.N.g^k.\log k)$ et celle en espace est $O(n.N.g^k)$.

Les cliques obtenues ne sont bien sûr pas nécessairement maximales. La chaîne est en effet de longueur finie et certains symboles appartenant à la différence de deux cliques maximales de R peuvent ne jamais y être présents. Si l'on souhaite travailler avec des blocs de mots qui représentent de telles cliques maximales, il faut encore, après le tri et la séparation de la liste $\mathcal{L}'$ suivant la valeur du premier champ, effectuer des tests d'inclusion des ensembles de positions obtenus (correspondant aux seconds champs des éléments de $\mathcal{L}'$) afin d'éliminer les cliques incluses dans une autre. Pour chaque étape k, cette opération peut se faire en $O(n.N.k.g^{2k})$ opérations [Soldano *et al.*, 1995], toujours dans le cas de $N \geq 1$ chaînes. Les étiquettes correspondant à des cliques non-maximales sont exclues de $\mathcal{L}'$ et la numérotation procède alors uniquement sur les cliques maximales restantes. L'algorithme complet s'appelle KMR Cliques abrégé en KMRC [Soldano *et al.*, 1995].

Nous reviendrons sur KMR et KMRC dans le chapitre 5, vus cette fois comme des cas spéciaux des méthodes de comparaisons indirectes de chaînes. Ces algorithmes seront alors abordés sous deux angles un peu différents de celui présenté ici mais qui y apparaissent déjà en filigrane : l'angle partitionnement/raffinement d'ensembles (de positions) et l'angle parcours dans un arbre. Notons pour clore cette section que, comme dans le cas de MACAW ou des algorithmes de la section 4.2.3.2, les blocs de mots communs identiques ou similaires obtenus par KMR ou KMRC peuvent être utilisés comme points d'ancrage pour un alignement multiple des chaînes. Un positionnement optimal des blocs est un problème NP-complet et seules des heuristiques ont été proposées à ce jour [Landraud *et al.*, 1989] [Viari and Pothier, 1993].

4.2.3.5 Le cas des structures

Ainsi que nous l'avons vu dans la section 3.2.3.2.2, il est possible d'établir une relation entre monomères en tant qu'objets 3D cette fois. Cette relation possède les mêmes caractéristiques que celle portant sur les monomères en tant que symboless, simplement elle intervient sur un alphabet plus grand et est définie de manière différente, sur une notion de voisinage de carrés dans un plan correspondant à une équivalence entre paires d'angles (les coordonnées internes de la macromolécule). Une version de KMRC adaptée à la comparaison multiple de structures de protéines codées en chaînes de caractères a ainsi été réalisée [Viari, 1995]. Une innovation a cependant été effectuée dans l'algorithme, qu'il est intéressant de noter au passage car elle introduit l'idée d'un 'mot spécial', composé de symboles non contigus.

L'introduction de cette innovation a été nécessaire à cause d'une particularité des objets à comparer (la structure des protéines) qui rendait parfois l'approche 'normale' de KMRC inefficace en pratique. KMRC en effet traite les structures comme des chaînes et donc les motifs structuraux comme des suites contiguës de caractères. Or la structure d'une protéine possède, ainsi que nous l'avons vu dans le chapitre 2, un certain nombre de régularités locales associées à ce qui est appelé sa structure secondaire. En particulier, elle est composée d'un nombre plus ou moins grand d'hélices. En termes de coordonnées internes, ces dernières sont représentées par des suites contiguës de paires d'angles approximativement identiques, et sont donc codées par des symboles identiques dans le nouvel alphabet obtenu par échantillonnage de la carte de Ramachandran (voir la section 3.2.2.3). Les chaînes de caractères correspondant à ces structures peuvent ainsi présenter de nombreux mots (de longueur entre 4 et 20 en général) composés de la répétition d'un même symbole. Chercher les mots communs à un ensemble de telles chaînes en les repérant par longueurs croissantes à la manière KMR ou KMRC peut ainsi consommer beaucoup de temps et d'espace en mémoire puisque typiquement ces chaînes peuvent avoir un très grand nombre de mots en relation par R_k pour des valeurs petites de k (autour de 4-5). Ces mots correspondent le plus souvent à de petites hélices (ou de petits morceaux d'hélices) et ne sont en général pas très intéressants. L'idéal était donc de réussir à les éviter, tout en ne perdant pas l'information nécessaire pour identifier tous les motifs structuraux plus longs (grandes hélices ou autres structures). L'idée a consisté à travailler, au moins au départ, non avec des 'vrais mots', c'est-à-dire, avec des mots composés de suites contiguës de symboles, mais avec des 'mots spéciaux'. Par exemple, nous pouvons définir un 'mot spécial' de longueur 2 comme étant un 'vrai mot' de longueur 8 dont seules les première et dernière positions 'comptent'. Deux 'mots spéciaux' de longueur 2 sont alors en relation par R_2 si et seulement si leurs premier et dernier symboles sont en relation par R. L'algorithme recherche ainsi les mots répétés en 'sautant' le long de la chaîne (d'où son nom de KMRS — S comme 'Saut' [Sagot *et al.*, 1995b]) jusqu'à ce qu'une certaine longueur k_0 soit atteinte à partir de laquelle les mots redeviennent 'normaux' et l'algorithme KMRS exactement pareil à KMRC. La complexité en temps finale de KMRS est en $O(n.N.k.(k_0 + \log(k - k_0).g^{2k})$, celle en espace demeure la même qu'indiqué ci-dessus.

4.2.3.6 Recherche de mots communs avec erreurs — une transition vers les méthodes indirectes

Mentionnons enfin un algorithme qui ne peut être classé ni parmi les méthodes de recherche de mots communs utilisant un arbre de suffixes, ni parmi celles de comparaison indirecte bien qu'il soit apparenté aux deux. L'algorithme de Lefevre [Lefevre and Ikeda, 1993a] [Lefevre and Ikeda, 1993b] [Lefevre and Ikeda, 1994] se sert en effet d'un arbre des suffixes (l'arbre des positions de Weiner, moins compact que celui de McCreight) pour repérer des blocs de mots similaires cette fois et non plus uniquement identiques ou équivalents, ce qui veut dire qu'un certain nombre de substitutions (pas de trous) est permis entre les mots. L'arbre n'est cependant utilisé que pour produire un index exact de tous les mots dans les chaînes et les blocs sont ensuite formés simplement en se servant de l'arbre pour comparer tous ces mots entre eux. Chaque bloc est ainsi composé de tous les mots se trouvant à une certaine distance de Hamming d'un mot présent dans une des chaînes (dans les méthodes indirectes, les objets contre lesquels les mots dans les chaînes sont comparés sont externes à ces dernières et peuvent ne même pas être des mots — voir la section 3.3.3 du chapitre 3).

Le fait que des substitutions soient autorisées signifie que la comparaison d'un mot avec l'arbre donne lieu à plusieurs parcours dans celui-ci. Si la longueur des blocs identifiés est fixe, notons-la m, et si aucune substitution n'est permise, la complexité de l'algorithme est $O(n.N.m)$. Si l'on recherche les blocs similaires de longueur maximale, toutes les longueurs possibles de blocs (satisfaisant une contrainte de quorum allant de 2 à N) doivent être traitées au préalable et la complexité devient alors $O(n^3.N)$ dans le pire des cas, toujours sans erreurs. Enfin si e substitutions sont autorisées, la complexité en théorie est $O(e.n.N.m)$ ou $O(e.n^3.N)$ respectivement (étant donné un motif u de longueur m, il faut $O(e.m)$ opérations en utilisant un arbre des suffixes pour déterminer s'il existe un mot v d'un ensemble de chaînes tel que la distance de Hamming entre u et v est au plus de e [Gusfield, 1995]). La complexité en espace est toujours $O(n.N)$.